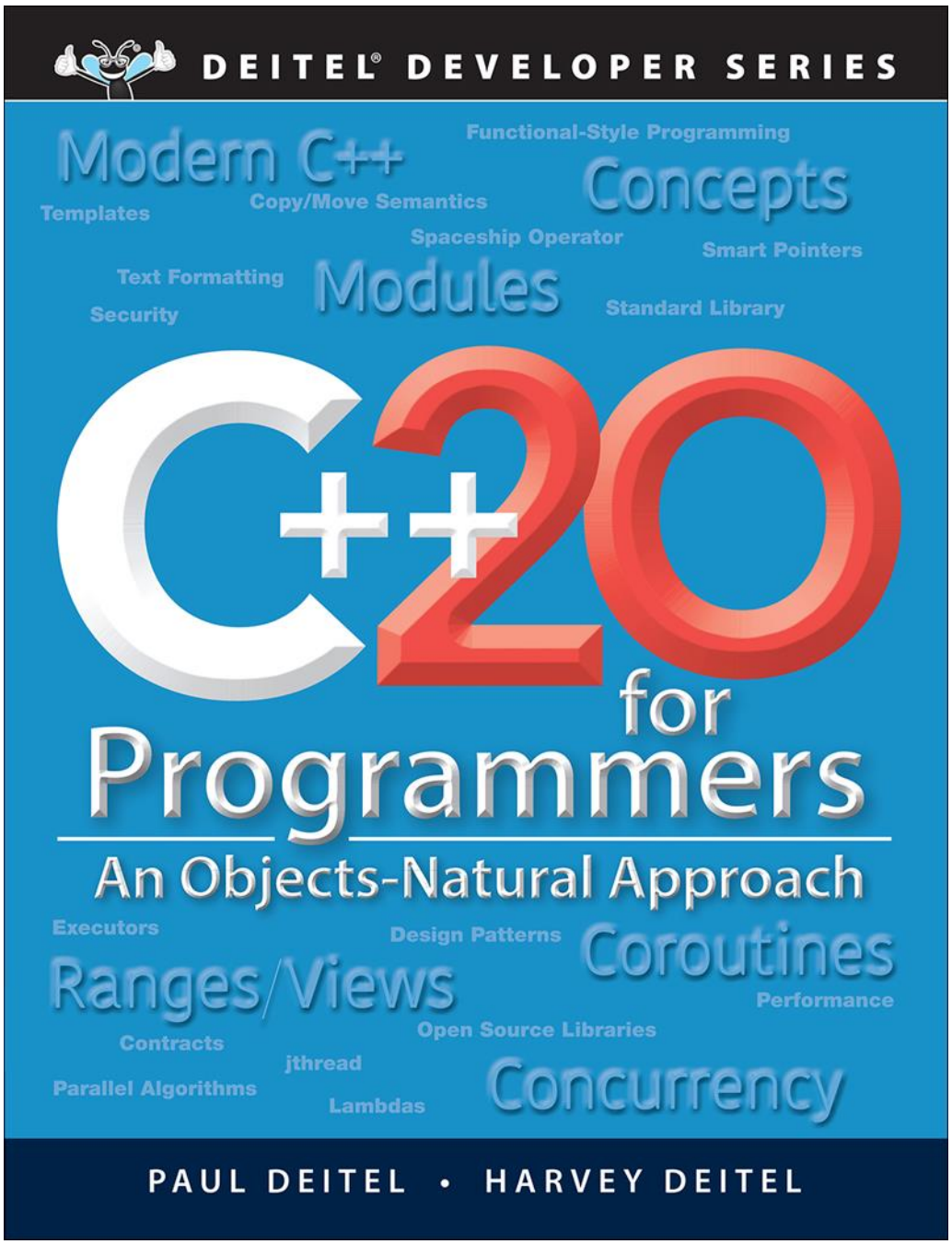


Copyright © 2022 Pearson Education, Inc.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit https://www.pearson.com/permissions/.
Deitel and the double-thumbs-up bug are registered trademarks of Deitel & Associates, Inc.

Cover design by Paul Deitel, Harvey Deitel, and Chuti Prasertsith

ISBN-13: 978-0-13-690569-1
ISBN-10: 0-13-690569-2

Contents

Preface ... 13

P.1 Modern C++ ... 13

P.2 Target Audiences ... 14

P.3 Live-Code Approach and Getting the Code ... 14

P.4 Three Industrial-Strength Compilers ... 14

P.6 “Objects-Natural” Learning Approach ... 15

P.8 How to Get the Online Chapters and Appendices ... 16

P.9 C++ Core Guidelines ... 17

P.11 Teaching Approach ... 18

P.13 Docker ... 18

P.14 Some Key C++ Documentation and Resources ... 19

Before You Begin ... 21

Compilers We Use in *C++20 for Programmers* ... 21

Some Examples Do Not Compile and Run on All Three Compilers ... 21

Installing Visual Studio Community Edition on Windows ... 21

Installing Xcode on macOS ... 22

Installing the Most Recent GNU C++ Version ... 22

Installing the GNU Compiler Collection in Ubuntu Linux Running on the Windows Subsystem for Linux ... 22

Docker and Docker Containers ... 23

A Note Regarding the {fmt} Text-Formatting Library ... 24

(Industrial-Strength) **Static Code Analysis Tools** ... 25

1. Intro and Test-Driving Popular, Free C++ Compilers ... 26

1.1 Introduction ... 26

1.2 Test-Driving a C++20 Application ... 28

1.3 Moore’s Law, Multi-Core Processors and Concurrent Programming ... 41

1.4 A Brief Refresher on Object Orientation ... 43

1.5 Wrap-Up ... 45

2. Intro to C++20 Programming ... 46

2.1 Introduction ... 46

2.2 First Program in C++: Displaying a Line of Text ... 47

2.3 Modifying Our First C++ Program ... 50

2.4 Another C++ Program: Adding Integers ... 51

2.5 Arithmetic ... 56

2.6 Decision Making: Equality and Relational Operators ... 57

2.7 Objects Natural: Creating and Using Objects of Standard-Library Class `string` ... 61

2.8 Wrap-Up ... 64

3. Control Statements: Part 1 66
3.1 Introduction 67
3.2 Control Structures 67
3.3 `if` Single-Selection Statement 70
3.4 `if` ...`else` Double-Selection Statement 71
3.5 `while` Iteration Statement 74
3.6 Counter-Controlled Iteration 75
3.7 Sentinel-Controlled Iteration 77
3.8 Nested Control Statements 81
3.9 Compound Assignment Operators 84
3.10 Increment and Decrement Operators 85
3.11 Fundamental Types Are Not Portable 88
3.12 Objects Natural Case Study: Arbitrary-Sized Integers 89
3.13 C++20: Text Formatting with Function **`format`** 93
3.14 Wrap-Up 95
4. Control Statements, Part 2 96
4.1 Introduction 96
4.2 Essentials of Counter-Controlled Iteration 97
4.3 `for` **Iteration Statement** 98
4.4 Examples Using the `for` Statement 100
4.5 Application: Summing Even Integers 101
4.6 Application: Compound-Interest Calculations 101
4.7 `do`... while Iteration Statement 105
4.8 `switch` Multiple-Selection Statement 106
17 4.9 C++17 Selection Statements with Initializers 111
4.10 `break` and **`continue`** Statements 113
4.11 Logical Operators 114
Err 4.12 Confusing the Equality (`==`) and Assignment (`=`) Operators 119
4.13 Objects-Natural Case Study: Using the **`miniz-cpp`** Library to Write and Read ZIP files[7] 120
20 4.14 C++20 Text Formatting with Field Widths and Precisions 125
4.15 Wrap-Up 127
5. Functions and an Intro to Function Templates 128
5.1 Introduction 129
5.2 C++ Program Components 129
5.3 Math Library Functions 130
5.4 Function Definitions and Function Prototypes 134

5.5 Order of Evaluation of a Function's Arguments ... 136
5.6 Function-Prototype and Argument-Coercion Notes ... 136
5.7 C++ Standard Library Headers ... 139
5.8 Case Study: Random-Number Generation ... 143
5.9 Case Study: Game of Chance; Introducing Scoped `enum`s ... 148
5.10 Scope Rules ... 153
5.11 Inline Functions ... 157
5.12 References and Reference Parameters ... 158
5.13 Default Arguments ... 161
5.14 Unary Scope Resolution Operator ... 163
5.15 Function Overloading ... 163
5.16 Function Templates ... 166
5.17 Recursion ... 169
5.18 Example Using Recursion: Fibonacci Series ... 172
5.19 Recursion vs. Iteration ... 175
5.20 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz ... 177
5.21 Wrap-Up ... 180
6. `array`s, `vector`s, Ranges and Functional-Style Programming ... 182
6.1 Introduction ... 182
6.2 `array`s ... 183
6.3 Declaring `array`s ... 184
6.4 Initializing `array` Elements in a Loop ... 184
6.5 Initializing an `array` with an Initializer List ... 186
6.6 C++11 Range-Based for and C++20 Range-Based for with Initializer ... 188
6.7 Calculating `array` Element Values and an Intro to constexpr ... 190
6.8 Totaling `array` Elements ... 191
6.9 Using a Primitive Bar Chart to Display array Data Graphically ... 192
6.10 Using array Elements as Counters ... 193
6.11 Using arrays to Summarize Survey Results ... 194
6.12 Sorting and Searching `array`s ... 195
6.13 Multidimensional `array`s ... 197
6.14 Intro to Functional-Style Programming ... 201
6.15 Objects-Natural Case Study: C++ Standard Library Class Template `vector` ... 207
6.16 Wrap-Up ... 213
7. (Downplaying) Pointers in Modern C++ ... 214
7.1 Introduction ... 215
7.2 Pointer Variable Declarations and Initialization ... 216

7.3 Pointer Operators ... 217
7.4 Pass-by-Reference with Pointers ... 220
7.5 Built-In Arrays ... 224
20 7.6 Using C++20 **`to_array`** to Convert a Built-in Array to a **`std::array`** ... 226
7.7 Using **`const`** with Pointers and the Data Pointed To ... 227
7.8 sizeof Operator ... 230
7.9 Pointer Expressions and Pointer Arithmetic ... 232
20 7.10 Objects-Natural Case Study: C++20 **`span`**s—Views of Contiguous Container Elements ... 235
7.11 A Brief Intro to Pointer-Based Strings ... 240
7.12 Looking Ahead to Other Pointer Topics ... 244
7.13 Wrap-Up ... 244
8. `string`s, `string_view`s, Text Files, CSV Files and Regex ... 246
8.1 Introduction ... 247
8.2 **`string`** Assignment and Concatenation ... 247
8.3 Comparing **`string`**s ... 249
8.4 Substrings ... 251
8.5 Swapping **`string`**s ... 251
8.6 **`string`** Characteristics ... 252
8.7 Finding Substrings and Characters in a **`string`** ... 254
8.8 Replacing and Erasing Characters in a **`string`** ... 256
8.9 Inserting Characters into a **`string`** ... 258
8.10 C++11 Numeric Conversions ... 259
17 8.11 C++17 **`string_view`** ... 260
8.12 Files and Streams ... 263
8.13 Creating a Sequential File ... 264
8.14 Reading Data from a Sequential File ... 267
14 8.15 C++14 Reading and Writing Quoted Text ... 269
8.16 Updating Sequential Files ... 270
8.17 String Stream Processing ... 271
8.18 Raw String Literals ... 273
8.19 Objects-Natural Case Study: Reading and Analyzing a CSV File Containing ***`Titanic`*** Disaster Data ... 274
8.20 Objects-Natural Case Study: Intro to Regular Expressions ... 284
8.21 Wrap-Up ... 291
9. Custom Classes ... 292
9.1 Introduction[1] ... 293
9.2 Test-Driving an **`Account`** Object ... 293
9.3 **`Account`** Class with a Data Member and *Set* and *Get* Member Functions ... 295

9.4 `Account` Class: Custom Constructors ... 298
9.5 Software Engineering with `Set` and `Get` Member Functions ... 302
9.6 `Account` Class with a Balance ... 303
9.7 `Time` Class Case Study: Separating Interface from Implementation ... 306
9.8 Compilation and Linking Process ... 313
9.9 Class Scope and Accessing Class Members ... 314
9.10 Access Functions and Utility Functions ... 315
9.11 Time Class Case Study: Constructors with Default Arguments ... 315
9.12 Destructors ... 321
9.13 When Constructors and Destructors Are Called ... 321
9.14 `Time` Class Case Study: A Subtle Trap —Returning a Reference or a Pointer to a `private` Data Member ... 324
9.15 Default Assignment Operator ... 326
9.16 `const` Objects and `const` Member Functions ... 328
9.17 Composition: Objects as Members of Classes ... 330
9.18 `friend` Functions and `friend` Classes ... 334
9.19 The `this` Pointer ... 336
9.20 `static` Class Members: Classwide Data and Member Functions ... 341
20 9.21 Aggregates in C++20 ... 345
9.22 Objects-Natural Case Study: Serialization with JSON ... 347
9.23 Wrap-Up ... 353
10. OOP: Inheritance and Runtime Polymorphism ... 355
10.1 Introduction ... 356
10.2 Base Classes and Derived Classes ... 358
10.3 Relationship Between Base and Derived Classes ... 361
10.4 Constructors and Destructors in Derived Classes ... 369
10.5 Intro to Runtime Polymorphism: Polymorphic Video Game ... 370
10.6 Relationships Among Objects in an Inheritance Hierarchy ... 371
10.7 Virtual Functions and Virtual Destructors ... 376
10.8 Abstract Classes and Pure `virtual` Functions ... 381
10.9 Case Study: Payroll System Using Runtime Polymorphism ... 383
10.10 Runtime Polymorphism, Virtual Functions and Dynamic Binding “Under the Hood” ... 392
10.11 Non-Virtual Interface (NVI) Idiom ... 395
10.12 Program to an Interface, Not an Implementation[24] ... 401
10.13 Runtime Polymorphism with `std::variant` and `std::visit` ... 408
10.14 Multiple Inheritance ... 414
10.15 `protected` Class Members: A Deeper Look ... 422

10.16 `public`, `protected` and `private` Inheritance ... 423
10.17 More Runtime Polymorphism Techniques; Compile-Time Polymorphism ... 424
10.18 Wrap-Up ... 428
11. Operator Overloading, Copy/Move Semantics and Smart Pointers ... 430
11.1 Introduction ... 431
11.2 Using the Overloaded Operators of Standard Library Class `string` ... 432
11.3 Operator Overloading Fundamentals ... 437
11.4 (Downplaying) Dynamic Memory Management with `new` and `delete` ... 439
11.5 Modern C++ Dynamic Memory Management: RAII and Smart Pointers ... 441
11.6 `MyArray` Case Study: Crafting a Valuable Class with Operator Overloading ... 444
20 11.7 C++20 Three-Way Comparison Operator (`<=>`) ... 474
11.8 Converting Between Types ... 477
11.9 `explicit` Constructors and Conversion Operators ... 477
11.10 Overloading the Function Call Operator () ... 480
11.11 Wrap-Up ... 480
12. Exceptions and a Look Forward to Contracts ... 482
12.1 Introduction ... 483
12.2 Exception-Handling Flow of Control ... 486
12.3 Exception Safety Guarantees and noexcept ... 491
12.4 Rethrowing an Exception ... 492
12.5 Stack Unwinding and Uncaught Exceptions ... 494
12.6 When to Use Exception Handling ... 496
12.7 Constructors, Destructors and Exception Handling ... 499
12.8 Processing new Failures ... 502
12.9 Standard Library Exception Hierarchy ... 505
12.10 C++'s Alternative to the finally Block: Resource Acquisition Is Initialization (RAII) ... 507
12.11 Some Libraries Support Both Exceptions and Error Codes ... 507
12.12 Logging ... 508
12.13 Looking Ahead to Contracts[49] ... 509
12.14 Wrap-Up ... 517
13. Standard Library Containers and Iterators ... 519
13.1 Introduction ... 520
13.2 Introduction to Containers[3] ... 522
13.3 Working with Iterators ... 528
13.4 A Brief Introduction to Algorithms ... 534
13.5 Sequence Containers ... 534
13.6 `vector` Sequence Container ... 535

13.7 `list` Sequence Container ... 542
13.8 `deque` Sequence Container ... 547
13.9 Associative Containers ... 548
13.10 Container Adaptors ... 557
13.11 `bitset` Near Container ... 561
13.12 Optional: A Brief Intro to Big *O* ... 563
13.13 Optional: A Brief Intro to Hash Tables ... 566
13.14 Wrap-Up ... 568
14. Standard Library Algorithms and C++20 Ranges & Views ... 569
14.1 Introduction ... 570
14.2 Algorithm Requirements: C++20 Concepts ... 571
14.3 Lambdas and Algorithms ... 574
14.4 Algorithms ... 577
14.5 Function Objects (Functors) ... 615
14.6 Projections ... 620
20 14.7 C++20 Views and Functional-Style Programming ... 622
14.8 Intro to Parallel Algorithms ... 629
14.9 Standard Library Algorithm Summary ... 630
14.10 A Look Ahead to C++23 Ranges ... 633
14.11 Wrap-Up ... 634
15. Templates, C++20 Concepts and Metaprogramming ... 636
15.1 Introduction ... 637
15.2 Custom Class Templates and Compile-Time Polymorphism ... 640
20 15.3 C++20 Function Template Enhancements ... 645
20 15.4 C++20 **Concepts: A First Look** ... 647
15.5 Type Traits ... 656
20 15.6 C++20 Concepts: A Deeper Look ... 661
15.7 Testing C++20 Concepts with static_assert ... 672
15.8 Creating a Custom Algorithm ... 674
15.9 Creating a Custom Container and Iterators ... 676
15.10 Default Arguments for Template Type Parameters ... 690
15.11 Variable Templates ... 691
15.12 Variadic Templates and Fold Expressions ... 691
15.13 Template Metaprogramming ... 706
15.14 Wrap-Up ... 717
16. C++20 Modules: Large-Scale Development ... 718
16.1 Introduction ... 719

16.2 Compilation and Linking Before C++20 ... 721
16.3 Advantages and Goals of Modules ... 722
16.4 Example: Transitioning to Modules—Header Units ... 723
16.5 Modules Can Reduce Translation Unit Sizes and Compilation Times ... 726
16.6 Example: Creating and Using a Module ... 727
16.7 Global Module Fragment ... 735
16.8 Separating Interface from Implementation ... 736
16.9 Partitions ... 743
16.10 Additional Modules Examples ... 751
16.11 Migrating Code to Modules ... 757
16.12 Future of Modules and Modules Tooling ... 757
16.13 Wrap-Up ... 759
17. Parallel Algorithms and Concurrency: A High-Level View ... 765
17.1 Introduction ... 766
17 17.2 Standard Library Parallel Algorithms (C++17) ... 768
17.3 Multithreaded Programming ... 776
17.4 Launching Tasks with `std::jthread` ... 781
17.5 Producer–Consumer Relationship: A First Attempt ... 785
17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data ... 792
17.7 Producer–Consumer: Minimizing Waits with a Circular Buffer ... 804
17.8 Readers and Writers ... 812
17.9 Cooperatively Canceling `jthread`s ... 813
17.10 Launching Tasks with `std::async` ... 816
17.11 Thread-Safe, One-Time Initialization ... 823
17.12 A Brief Introduction to Atomics ... 824
20 17.13 Coordinating Threads with C++20 Latches and Barriers ... 827
20 17.14 C++20 Semaphores ... 834
17.15 C++23: A Look to the Future of C++ Concurrency ... 837
17.16 Wrap-Up ... 838
18. C++20 Coroutines ... 840
18.1 Introduction ... 840
18.2 Coroutine Support Libraries ... 841
18.3 Installing the `concurrencpp` and `generator` Libraries ... 843
18.4 Creating a Generator Coroutine with `co_yield` and the `generator` Library ... 843
18.5 Launching Tasks with `concurrencpp` ... 847
18.6 Creating a Coroutine with `co_await` and `co_return` ... 851
18.7 Low-Level Coroutines Concepts ... 858

18.8 C++23 Coroutines Enhancements ... 861
18.9 Wrap-Up ... 861
19. Stream I/O & C++20 Text Formatting ... 862
19.1 Introduction ... 863
19.2 Streams ... 863
19.3 Stream Output ... 865
19.4 Stream Input ... 866
19.5 Unformatted I/O Using `read`, `write` and `gcount` ... 870
19.6 Stream Manipulators ... 871
19.7 Stream Error States ... 884
19.8 Tying an Output Stream to an Input Stream ... 887
20 19.9 C++20 Text Formatting ... 887
19.10 Wrap-Up ... 892
20. Other Topics and a Look Toward the Future of C++ ... 894
20.1 Introduction ... 895
20.2 `const_cast` Operator ... 895
20.3 Storage Classes and Storage Duration ... 897
20.4 `namespaces` ... 900
20.5 Operator Keywords ... 904
20.6 Pointers to Class Members (`.*` and `->*`) ... 906
20.7 Case Study: Payroll System Using Polymorphism and Runtime Type Information — Downcasting, `dynamic_cast`, `typeid` and `type_info` ... 907
11 20.8 Inheriting Base-Class Constructors ... 910
17 20.9 C++17 and C++20: `[[nodiscard]]` Attribute ... 911
20.10 **`shared_ptr` and `weak_ptr` Smart Pointers** ... 912
11 20.11 Trailing Return Types for Functions ... 923
20.12 `decltype` ... 923
20.13 **`initializer_list` Class Template** ... 924
20 20.14 C++20: `[[likely]]` **and** `[[unlikely]]` **Attributes** ... 925
23 20.15 A Look Toward C++23 ... 926
20.16 Wrap-Up ... 928
20.17 Closing Notes ... 929
A. Operator Precedence and Grouping ... 930
B. Character Set ... 934
C. Number Systems ... 935
C.1 Introduction ... 935
C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers ... 939

C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers ... 941
C.4 Converting from Binary, Octal or Hexadecimal to Decimal ... 941
C.5 Converting from Decimal to Binary, Octal or Hexadecimal ... 943
C.6 Negative Binary Numbers: Two's Complement Notation ... 944
D. Preprocessor ... 947
D.1 Introduction ... 947
D.2 `#include` Preprocessing Directive ... 947
D.3 `#define` Preprocessing Directive: Symbolic Constants ... 948
D.4 `#define` Preprocessing Directive: Macros ... 948
D.5 Conditional Compilation ... 951
D.6 `#error` and `#pragma` Preprocessing Directives ... 953
D.7 Operators # and ## ... 953
D.8 Predefined Symbolic Constants ... 954
D.9 Assertions ... 954
E. Bit Manipulation ... 956
E.1 Introduction ... 956
E.2 Bitwise Operators ... 956
E.3 Bit Fields ... 964
Index ... 968
Symbols ... 968
A ... 968
B ... 970
C ... 971
D ... 978
E ... 980
F ... 981
G ... 983
H ... 984
I ... 984
J ... 986
K ... 987
L ... 987
M ... 988
N ... 990
O ... 991
P ... 993
Q ... 995

R 995
S 997
T 1002
U 1003
V 1004
W 1005
X 1005
Y 1005
Z 1005

Preface

Welcome to ***C++20 for Programmers: An Objects-Natural Approach***. This book presents leading-edge computing technologies for software developers. It conforms to the C++20 standard (1,834 pages), which the ISO C++ Standards Committee approved in September 2020.[1,2]

1. The final draft C++ standard is located at: https://timsong-cpp.github.io/cppwp/n4861/. This version is free. The published final version (ISO/IEC 14882:2020) may be purchased at https://www.iso.org/standard/79358.html.
2. Herb Sutter, "C++20 Approved, C++23 Meetings and Schedule Update," September 6, 2020. Accessed January 11, 2022. https://herbsutter.com/2020/09/06/c20-approved-c23-meetings-and-schedule-update/.

The C++ programming language is popular for building high-performance business-critical and mission-critical computing systems—operating systems, real-time systems, embedded systems, game systems, banking systems, air-traffic-control systems, communications systems and more. This book is an introductory- through intermediate-level tutorial presentation of the C++20 version of C++, which is among the world's most popular programming languages,[3] and its associated standard libraries. We present a friendly, contemporary, code-intensive, case-study-oriented introduction to C++20. In this Preface, we explore the "soul of the book."

3. Tiobe Index for January 2022. Accessed January 7, 2022. http://www.tiobe.com/tiobe-index.

P.1 Modern C++

We focus on **Modern C++**, which includes the four most recent C++ standards—**C++20**, **C++17**, **C++14** and **C++11**, with a look toward key features anticipated for **C++23** and later. A common theme of this book is to focus on the new and improved ways to code in C++. We employ best practices, emphasizing current professional software-development Modern C++ idioms, and we focus on performance, security and software engineering issues.

Keep It Topical

"*Who dares to teach must never cease to learn.*"[4] (J. C. Dana)

4. John Cotton Dana. From https://www.bartleby.com/73/1799.html: "In 1912 Dana, a Newark, New Jersey, librarian, was asked to supply a Latin quotation suitable for inscription on a new building at Newark State College (now Kean University), Union, New Jersey. Unable to find an appropriate quotation, Dana composed what became the college motto."—*The New York Times Book Review*, March 5, 1967, p. 55."

To "take the pulse" of Modern C++, which changes the way developers write C++ programs, we read, browsed or watched approximately 6,000 current articles, research papers, white papers, documentation pieces, blog posts, forum posts and videos.

C++ Versions

As a developer, you might work on C++ legacy code or projects requiring specific C++ versions. So, we use margin icons like the **"20" icon** shown here to mark each mention of a Modern C++ language feature with the C++ version in which it first appeared. The icons help you see C++ evolving, often from programming with low-level details to easier-to-use, higher-level forms of expression. These

trends help reduce development times, and enhance performance, security and system maintainability.

P.2 Target Audiences

C++20 for Programmers: An Objects-Natural Approach has several target audiences:

- C++ software developers who want to learn the latest C++20 features in the context of a full-language, professional-style tutorial,
- non-C++ software developers who are preparing to do a C++ project and want to learn the latest version of C++,
- software developers who learned C++ in college or used it professionally some time ago and want to refresh their C++ knowledge in the context of C++20, and
- professional C++ trainers developing C++20 courses.

P.3 Live-Code Approach and Getting the Code

At the heart of the book is the Deitel signature **live-code approach**. Rather than code snippets, we show C++ as it's intended to be used in the context of hundreds of complete, working, real-world C++ programs with live outputs.

Read the **Before You Begin** section that follows this Preface to learn how to set up your **Windows**, **macOS** or **Linux** computer to run the 200+ code examples consisting of approximately 15,000 lines of code. All the source code is available free for download at

- https://github.com/pdeitel/CPlusPlus20ForProgrammers
- https://www.deitel.com/books/c-plus-plus-20-for-programmers
- https://informit.com/title/9780136905691 (see Section P.8)

For your convenience, we provide the book's examples in C++ source-code (`.cpp` and `.h`) files for use with integrated development environments and command-line compilers.

Note: If you're not familiar with Git and GitHub but are interested in learning about these essential developer tools, check out their guides at

```
https://guides.github.com/activities/hello-world/
```

See **Chapter 1's Test-Drives** (Section 1.2) for information on compiling and running the code examples with our three preferred compilers. Execute each program in parallel with reading the text to make your learning experience "come alive."

P.4 Three Industrial-Strength Compilers

We tested the code examples on the latest versions of

- **Visual C++®** in Microsoft® Visual Studio® Community edition on Windows®.,
- **Clang C++** (`clang++`) in Apple® Xcode® on macOS®, and in a Docker® container, and
- **GNU®C++** (`g++`) on Linux® and in the GNU Compiler Collection (GCC) Docker® container.

20 At the time of this writing, most C++20 features are fully implemented by all three compilers, some are implemented by a subset of the three and some are not yet implemented by any. We point out these differences as appropriate and will update our digital content as the compiler vendors

implement the remaining C++20 features. We'll also post code updates to the **book's GitHub repository**:

```
https://github.com/pdeitel/CPlusPlus20ForProgrammers
```

and both code and text updates on the book's websites:

```
https://www.deitel.com/books/c-plus-plus-20-for-programmers
https://informit.com/title/9780136905691
```

P.5 Programming Wisdom and Key C++20 Features

Throughout the book, we use margin icons to call your attention to **software-development wisdom** and **C++20 modules** and **concepts** features:

- SE **Software engineering observations** highlight architectural and design issues for proper software construction, especially for larger systems.
- Sec **Security best practices** help you strengthen your programs against attacks.
- Perf **Performance tips** highlight opportunities to make your programs run faster or minimize the amount of memory they occupy.
- Err **Common programming errors** help reduce the likelihood that you'll make the same mistakes.
- CG **C++ Core Guidelines** recommendations (introduced in Section P.9).
- Mod C++20's new **modules** features.
- Concepts C++20's new **concepts** features.

P.6 "Objects-Natural" Learning Approach

In Chapter 9, we'll cover how to develop **custom C++20 classes**, then continue our treatment of object-oriented programming throughout the rest of the book.

What Is Objects Natural?

In the early chapters, you'll work with**preexisting classes that do significant things**. You'll quickly create objects of those classes and get them to "strut their stuff" with a minimal number of simple C++ statements. We call this the **"Objects-Natural Approach."**

Given the massive numbers of free, open-source class libraries created by the C++ community, **you'll be able to perform powerful tasks long before you study how to create your own custom C++ classes in Chapter 9**. This is one of the most compelling aspects of working with object-oriented languages, in general, and with a mature object-oriented language like C++, in particular.

Free Classes

We emphasize using the huge number of valuable free classes available in the C++ ecosystem. These typically come from:

- the C++ Standard Library,
- platform-specific libraries, such as those provided with Microsoft Windows, Apple macOS or various Linux versions,
- free third-party C++ libraries, often created by the open-source community, and
- fellow developers, such as those in your organization.

We encourage you to view lots of free, open-source C++ code examples (available on sites such as GitHub) for inspiration.

The Boost Project

Boost provides 168 open-source C++ libraries.[5] It also serves as a "breeding ground" for new capabilities that are eventually incorporated into the C++ standard libraries. Some that have been added to Modern C++ include multithreading, random-number generation, smart pointers, tuples, regular expressions, file systems and `string_view`s.[6] The following StackOverflow answer lists Modern C++ libraries and language features that evolved from the Boost libraries:[7]

5. "Boost 1.78.0 Library Documentation." Accessed January 9, 2022. https://www.boost.org/doc/libs/1_78_0/.
6. "Boost C++ Libraries." Wikipedia. Wikimedia Foundation. Accessed January 9, 2022. https://en.wikipedia.org/wiki/Boost_(C%2B%2B_libraries).
7. Kennytm, Answer to "Which Boost Features Overlap with C++11?" Accessed January 9, 2022. https://stackoverflow.com/a/8852421.

```
https://stackoverflow.com/a/8852421
```

P.8 How to Get the Online Chapters and Appendices

We provide several **online chapters and appendices** on informit.com. Perform the following steps to register your copy of ***C++20 for Programmers: An Objects-Natural Approach*** on informit.com and access this online content:

1. Go to **https://informit.com/register** and sign in with an existing account or create a new one.
2. Under **Register a Product**, enter the ISBN **9780136905691**, then click **Submit**.
3. In your account page's **My Registered Products** section, click the **Access Bonus Content** link under ***C++20 for Programmers: An Objects-Natural Approach***.

This will take you to the book's online content page.

Online Chapters

20 **Chapter 19, Stream I/O; C++20 Text Formatting: A Deeper Look**, discusses standard C++ input/output capabilities and legacy formatting features of the `<iomanip>` library. We include these formatting features primarily for programmers who might encounter them in **legacy C++ code**. We also present **C++20's new text-formatting features** in more depth.

20 **Chapter 20, Other Topics**, presents miscellaneous C++ topics and looks forward to new features expected in C++23 and beyond.

Online Appendices

20 **Appendix C**, **Number Systems**, overviews the binary, octal, decimal and hexadecimal number systems.

Appendix D, **Preprocessor**, discusses additional features of the C++ preprocessor. Template metaprogramming (Chapter 15) and C++20 Modules (Chapter 16) obviate many of this appendix's features.

Appendix E, **Bit Manipulation**, discusses bitwise operators for manipulating the individual bits of integral operands and bit fields for compactly representing integer data.

P.9 C++ Core Guidelines

CG The **C++ Core Guidelines** (approximately 500 printed pages)

```
https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
```

are recommendations "to help people use modern C++ effectively."[11] They're edited by Bjarne Stroustrup (C++'s creator) and Herb Sutter (Convener of the ISO C++ Standards Committee). According to the overview:

11. C++ Core Guidelines, "Abstract." Accessed January 9, 2020. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-abstract.

SE Err Perf "*The guidelines are focused on relatively high-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast—you can afford to do things right.*"[12]

12. C++ Core Guidelines, "Abstract."

CG Throughout this book, we adhere to these guidelines as appropriate. You'll want to pay close attention to their wisdom. We point out many **C++ Core Guidelines** recommendations with a **CG icon**. There are hundreds of core guidelines divided into scores of categories and subcategories. Though this might seem overwhelming, static code analysis tools (Section Before you begin) can check your code against the guidelines.

Guidelines Support Library

The C++ Core Guidelines often refer to capabilities of the **Guidelines Support Library (GSL)**, which implements helper classes and functions to support various recommendations.[13] Microsoft provides an open-source GSL implementation on GitHub at

13. C++ Core Guidelines, "GSL: Guidelines Support Library." Accessed January 9, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-gsl.

```
https://github.com/Microsoft/GSL
```

We use GSL features in a few examples in the early chapters. Some GSL features have since been incorporated into the C++ standard library.

P.11 Teaching Approach

Sec ***C++20 for Programmers: An Objects-Natural Approach*** contains a rich collection of live-code examples. We stress program clarity and concentrate on building well-engineered software.

Using Fonts for Emphasis

We place the key terms and the index's page reference for each defining occurrence in **bold text** for easier reference. C++ code uses a `fixed-width font` (e.g., `x = 5`). We place on-screen components in the **bold Helvetica** font (e.g., the **File** menu).

Syntax Coloring

For readability, we syntax color all the code. In our e-books, our syntax-coloring conventions are as follows:

```
comments appear in green
keywords appear in dark blue
constants and literal values appear in light blue
errors appear in red
all other code appears in black
```

P.13 Docker

Sec We use **Docker**—a tool for packaging software into **containers** that bundle everything required to execute that software conveniently and portably across platforms. Some software packages require complicated setup and configuration. For many of these, you can download free preexisting Docker containers, avoiding complex installation issues. You can simply execute software locally on your desktop or notebook computers, making Docker a great way to help you get started with new technologies quickly, conveniently and economically.

We show how to install and execute Docker containers preconfigured with

- the GNU Compiler Collection (GCC), which includes the `g++` compiler, and
- the latest version of Clang's `clang++` compiler.

Each can run in **Docker** on **Windows**, **macOS** and **Linux**.

Docker also helps with **reproducibility**. Custom Docker containers can be configured with the software and libraries you use. This would enable others to recreate the environment you used, then reproduce your work, and will help you reproduce your own results. Reproducibility is especially important in the sciences and medicine—for example, when researchers want to prove and extend the work in published articles.

P.14 Some Key C++ Documentation and Resources

The book includes over 900 citations to videos, blog posts, articles and online documentation we studied while writing the manuscript. You may want to access some of these resources to investigate more advanced features and idioms. The website **cppreference.com** has become the defacto C++ documentation site. We reference it frequently so you can get more details about the standard C++ classes and functions we use throughout the book. We also frequently reference the final draft of the **C++20 standard document**, which is available for free on GitHub at

```
https://timsong-cpp.github.io/cppwp/n4861/
```

You may also find the following C++ resources helpful as you work through the book.

Documentation

- 20C++20 standard document final draft adopted by the C++ Standard Committee:

```
https://timsong-cpp.github.io/cppwp/n4861/
```

- C++ Reference at cppreference.com:

```
https://cppreference.com/
```

- Microsoft's C++ language documentation:

```
https://docs.microsoft.com/en-us/cpp/cpp/
```

- The GNU C++ Standard Library Reference Manual:

```
https://gcc.gnu.org/onlinedocs/libstdc++/manual/index.html
```

Blog:

- Sutter's Mill Blog—Herb Sutter on software development:

```
https://herbsutter.com/
```

- Microsoft's C++ Team Blog:

```
https://devblogs.microsoft.com/cppblog
```

- Marius Bancila's Blog:

```
https://mariusbancila.ro/blog/
```

- Jonathan Boccara's Blog:

```
https://www.fluentcpp.com/
```

- Bartlomiej Filipek's Blog:

```
https://www.cppstories.com/
```

- Rainer Grimm's Blog:

```
http://modernescpp.com/
```

- Arthur O'Dwyer's Blog:

```
https://quuxplusone.github.io/blog/
```

Additional Resources

- Bjarne Stroustrup's website:

```
https://stroustrup.com/
```

- Standard C++ Foundation website:

```
https://isocpp.org/
```

- C++ Standard Committee website:

```
http://www.open-std.org/jtc1/sc22/wg21/
```

Before You Begin

Compilers We Use in *C++20 for Programmers*

Before reading this book, ensure that you have a recent C++ compiler installed. We tested the code examples in *C++20 for Programmers* using the following free compilers:

- For Microsoft Windows, we used Microsoft Visual Studio Community edition, which includes the Visual C++ compiler and other Microsoft development tools.[1]

1. Visual Studio 2022 Community at the time of this writing.

- For macOS, we used the Apple Xcode[2] C++ compiler, which uses a version of the Clang C++ compiler.

2. Xcode 13.2.1 at the time of this writing.

- For Linux, we used the GNU C++ compiler[3]—part of the GNU Compiler Collection (GCC). GNU C++ is already installed on most Linux systems (though you might need to update the compiler to a more recent version) and can be installed on macOS and Windows systems.

3. GNU `g++` 11.2 at the time of this writing.

- You also can run the latest versions of GNU C++ and Clang C++ conveniently on Windows, macOS and Linux via Docker containers. See the "Docker and Docker Containers" section later in this Before You Begin section.

This Before You Begin describes installing the compilers and Docker. Section 1.2's test-drives demonstrate how to compile and run C++ programs using these compilers.

Some Examples Do Not Compile and Run on All Three Compilers

At the time of this writing (February 2022), the compiler vendors had not yet fully implemented some of C++20's new features. As those features become available, we'll retest the code, update our digital products and post updates for our print products at

```
https://deitel.com/cpp20fp
```

Installing Visual Studio Community Edition on Windows

If you are a Windows user, first ensure that your system meets the requirements for Microsoft Visual Studio Community edition at

```
https://docs.microsoft.com/en-us/visualstudio/releases/2022/system-requirements
```

Next, go to

```
https://visualstudio.microsoft.com/downloads/
```

Then perform the following installation steps:

1. Click **Free Download** under **Community**.
2. Depending on your web browser, you may see a pop-up at the bottom of your screen in which you can click **Run** to start the installation process. If not, double-click the installer file in your **Downloads** folder.
3. In the **User Account Control** dialog, click **Yes** to allow the installer to make changes to your system.
4. In the **Visual Studio Installer** dialog, click **Continue** to allow the installer to download the components it needs for you to configure your installation.
5. For this book's examples, select the option **Desktop Development with C++**, which includes the Visual C++ compiler and the C++ standard libraries.
6. Click **Install**. Depending on your Internet connection speed, the installation process can take a significant amount of time.

Installing Xcode on macOS

On macOS, perform the following steps to install Xcode:

1. Click the Apple menu and select **App Store...**, or click the **App Store** icon in the dock at the bottom of your Mac screen.
2. In the **App Store**'s **Search** field, type **Xcode**.
3. Click the **Get** button to install Xcode.

Installing the Most Recent GNU C++ Version

There are many Linux distributions, and they often use different software upgrade techniques. Check your distribution's online documentation for the proper way to upgrade GNU C++ to the latest version. You also can download GNU C++ for various platforms at

```
https://gcc.gnu.org/install/binaries.html
```

Installing the GNU Compiler Collection in Ubuntu Linux Running on the Windows Subsystem for Linux

You can install the GNU Compiler Collection on Windows via the Windows Subsystem for Linux (WSL), which enables you to run Linux in Windows. Ubuntu Linux provides an easy-to-use installer in the Windows Store, but first you must install WSL:

1. In the search box on your taskbar, type "Turn Windows features on or off," then click **Open** in the search results.
2. In the Windows Features dialog, locate **Windows Subsystem for Linux** and ensure that it is checked. If it is, WSL is already installed. Otherwise, check it and click **OK**. Windows will install WSL and ask you to reboot your system.

3. Once the system reboots and you log in, open the **Microsoft Store** app and search for **Ubuntu**, select the app named **Ubuntu** and click **Install**. This installs the latest version of Ubuntu Linux.
4. Once installed, click the **Launch** button to display the Ubuntu Linux command-line window, which will continue the installation process. You'll be asked to create a username and password for your Ubuntu installation—these do not need to match your Windows username and password.
5. When the Ubuntu installation completes, execute the following two commands to install the GCC and the GNU debugger—you may be asked enter your password for the account you created in Step 4:

```
sudo apt-get update
sudo apt-get install build-essential gdb
```

6. Confirm that `g++` is installed by executing the following command:

```
g++ --version
```

To access our code files, use the `cd` command change the folder within Ubuntu to:

```
cd /mnt/c/Users/YourUserName/Documents/examples
```

Use your own username and update the path to where you placed our examples on your system.

Docker and Docker Containers

Docker is a tool for packaging software into **containers** (also called **images**) that bundle *everything* required to execute that software across platforms, which is particularly useful for software packages with complicated setups and configurations. For many such packages, there are free preexisting Docker containers (often at https://hub.docker.com) that you can download and execute locally on your system. Docker is a great way to get started with new technologies quickly and conveniently. It is also a great way to experiment with new compiler versions.

Installing Docker

To use a Docker container, you must first install Docker. Windows and macOS users should download and run the **Docker Desktop** installer from

```
https://www.docker.com/get-started
```

Then follow the on-screen instructions. Also, sign up for a **Docker Hub** account on this webpage so you can take advantage of containers from https://hub.docker.com. Linux users should install **Docker Engine** from

```
https://docs.docker.com/engine/install/
```

Downloading the GNU Compiler Collection Docker Container

The GNU team maintains official Docker containers at

```
https://hub.docker.com/_/gcc
```

Once Docker is installed and running, open a Command Prompt[4] (Windows), Terminal (macOS/Linux) or shell (Linux), then execute the command

4. Windows users should choose **Run as administrator** when opening the Command Prompt.

```
docker pull gcc:latest
```

Docker downloads the GNU Compiler Collection (GCC) container's most current version (at the time of this writing, version 11.2). In one of Section 1.2's test-drives, we'll demonstrate how to execute the container and use it to compile and run C++ programs.

Downloading Clang Docker Container

Currently, the Clang team does not provide an official Docker container, but many working containers are available on https://hub.docker.com. For this book we used a popular one from

```
https://hub.docker.com/r/teeks99/clang-ubuntu
```

Open a Command Prompt (Windows), Terminal (macOS/Linux) or shell (Linux), then execute the command

```
docker pull teeks99/clang-ubuntu:latest
```

Docker downloads the Clang container's most current version (at the time of this writing, version 13). In one of Section 1.2's test-drives, we'll demonstrate how to execute the container and use it to compile and run C++ programs.

A Note Regarding the {fmt} Text-Formatting Library

Throughout the book many programs include the following line of code:

```
#include <fmt/format.h>
```

which enables our programs to use the open-source `{fmt}` library's text-formatting features.[5] Those programs include calls to the function `fmt::format`.

5. "`{fmt}`." Accessed February 15, 2022. https://github.com/fmtlib/fmt.

C++20's new text-formatting capabilities are a subset of the `{fmt}` library's features. In C++20, the preceding line of code should be

```
#include <format>
```

and the corresponding function calls should use the `std::format` function.

At the time of this writing, only Microsoft Visual C++ supported C++20's new text-formatting capabilities. For this reason, our examples use the open-source `{fmt}` library to ensure most of the examples will execute on all of our preferred compilers.

(Industrial-Strength) Static Code Analysis Tools

Err Sec **Static code analysis tools** let you quickly check your code for **common errors** and **security problems** and provide insights for code improvement. Using these tools is like having world-class experts checking your code. To help us adhere to the C++ Core Guidelines and improve our code in general, we used the following static-code analyzers:

- `clang-tidy`—https://clang.llvm.org/extra/clang-tidy/
- `cppcheck`—https://cppcheck.sourceforge.io/
- **Microsoft's C++ Core Guidelines static code analysis tools**, which are built into Visual Studio's static code analyzer

We used these three tools on the book's code examples to check for

- adherence to the C++ Core Guidelines,
- adherence to coding standards,
- adherence to modern C++ idioms,
- possible security problems,
- common bugs,
- possible performance issues,
- code readability
- and more.

Err We also used the compiler flag `-Wall` in the GNU `g++` and Clang C++ compilers to enable all compiler warnings. **With a few exceptions for warnings beyond this book's scope, we ensure that our programs compile without warning messages**. See the **Before You Begin** section for static analysis tool configuration information.

Installation

You can install `clang-tidy` on Linux with the following commands:

```
sudo apt-get update -y
sudo apt-get install -y clang-tidy
```

You can install `cppcheck` for various operating-system platforms by following the instructions at https://cppcheck.sourceforge.io/. For Visual C++, once you learn how to create a project in Section 1.2's test-drives, you can configure Microsoft's C++ Core Guidelines static code analysis tools as follows:

1. Right-click your project name in the **Solution Explorer** and select **Properties**.
2. In the dialog that appears, select **Code Analysis > General** in the left column, then set **Enable Code Analysis on Build** to **Yes** in the right column.
3. Next, select **Code Analysis > Microsoft** in the left column. Then, in the right column you can select a specific subset of the analysis rules in the drop-down list. We used the option **<Choose multiple rule sets...>** to select all the rule sets that begin with **C++ Core Check**. Click **Save As...**, give your custom rule set a name, click **Save**, then click **Apply**. (Note that this will produce large numbers of warnings for the `{fmt}` text-formatting library that we use in the book's examples.)

1. Intro and Test-Driving Popular, Free C++ Compilers

Objectives

In this chapter, you'll:

- Quickly get a "40,000-foot view" of this book's architecture and coverage of the large, complex and powerful programming language that is C++20.
- Test-drive compiling and running a C++ application using our three preferred compilers—Visual C++ in Microsoft Visual Studio on Windows, Clang C++ in Xcode on macOS and GNU `g++` on Linux.
- See how to execute Docker containers for the `g++` and `clang++` command-line compilers, so you can use these compilers on Windows, macOS or Linux.
- See resources where you can learn about C++'s history and milestones over 40+ years.
- Understand why concurrent programming is crucial in Modern C++ for getting maximum performance from today's multi-core processors.
- Review object-technology concepts used in the early chapters' objects-natural case studies and presented in the book's object-oriented programming chapters (starting with Chapter 9).

Outline

1.1 Introduction
1.2 Test-Driving a C++20 Application
1.2.1 Compiling and Running a C++20 Application with Visual Studio 2022 Community Edition on Windows
1.2.2 Compiling and Running a C++20 Application with Xcode on macOS
1.2.3 Compiling and Running a C++20 Application with GNU C++ on Linux
1.2.4 Compiling and Running a C++20 Application with **`g++`** in the GCC Docker Container
1.2.5 Compiling and Running a C++20 Application with **`clang++`** in a Docker Container
1.3 Moore's Law, Multi-Core Processors and Concurrent Programming
1.4 A Brief Refresher on Object Orientation
1.5 Wrap-Up

1.1 Introduction

20 Welcome to C++—one of the world's most popular programming languages.[1] We present Modern C++ in the context of C++20—the latest version standardized through the **International Organization for Standardization** (**ISO**). This chapter presents

1. "TIOBE Index." Accessed January 10, 2022. https://www.tiobe.com/tiobe-index/.

- a quick way for you to understand the book's superstructure,
- several test-drives of the most popular free C++ compilers,
- a discussion of Moore's law, multi-core processors and why concurrent programming is crucial to building high-performance applications in Modern C++, and
- a brief refresher on object-oriented programming concepts and terminology we'll use throughout the book.

•

This Book's Superstructure

Before you "dig in," we recommend that you get a "40,000-foot" view of the book's superstructure to understand where you're headed as you prepare to learn the large, complex and powerful language that is C++20. To do so, we recommend reviewing the following items:

- The one-page, full-color Table of Contents diagram inside the front cover provides a high-level overview of the book. You can view a scalable PDF version of this diagram at

```
https://deitel.com/cpp20fpTOCdiagram
```

- The back cover contains a concise introduction to the book, a bullet list of its key features and several testimonial comments. More are included on the inside back cover and its facing page. These comments are from the C++ subject-matter experts who reviewed the prepublication manuscript. Reading them will give you a nice overview of the book's features the reviewers felt were important. These comments are also posted on the book's webpage at

```
https://deitel.com/cpp20fp
```

- The **Preface** presents the "soul of the book" and our approach to Modern C++ programming. We introduce the "Objects-Natural Approach," in which you'll use small numbers of simple C++ statements to make powerful classes perform significant tasks—long before you create custom classes. Be sure to read the Tour of the Book, which points out the key features of each chapter. As you read the Tour, you might also want to refer to the Table of Contents diagram.

Resources on the History of C++

In 1979, Bjarne Stroustrup began creating C++, which he called "C with Classes."[2] There are now at least five million developers (with some estimates are as high as 7.5 million[3,4]) using C++ to build a wide range of business-critical and mission-critical systems and applications software.[5,6] Today's popular desktop operating systems—Windows[7] and macOS[8]—are partially written in C++. Many popular applications also are partially written in C++, including web browsers (e.g., Google Chrome[9] and Mozilla Firefox[10]), database management systems (e.g., MySQL[11]and MongoDB[12]) and more.

2. "Bjarne Stroustrup." Accessed January 10, 2022. https://en.wikipedia.org/wiki/Bjarne_Stroustrup.
3. "State of the Developer Nation, 21st Edition," Q3 2021. Accessed January 10, 2022. https://www.slashdata.co/free-resources/state-of-the-developer-nation-21st-edition.
4. Tim Anderson, "Report: World's Population of Developers Expands, Javascript Reigns, C# Overtakes PHP," April 26, 2021. Accessed January 10, 2022. https://www.theregister.com/2021/04/26/report_developers_slashdata/.
5. "Top 10 Reasons to Learn C++." Accessed January 10, 2022. https://www.geeksforgeeks.org/top-10-reasons-to-learn-c-plus-plus/.
6. "What Is C++ Used For? Top 12 Real-World Applications and Uses of C++." Accessed January 10, 2022. https://www.softwaretestinghelp.com/cpp-applications/.
7. "What Programming Language Is Windows Written In?" Accessed January 10, 2022. https://social.microsoft.com/Forums/en-US/65a1fe05-9c1d-48bf-bd40-148e6b3da9f1/what-programming-language-is-windows-written-in.
8. "macOS." Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. https://en.wikipedia.org/wiki/MacOS.

9. “Google Chrome.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. https://en.wikipedia.org/wiki/Google_Chrome.
10. “Firefox.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. https://en.wikipedia.org/wiki/Firefox.
11. “MySQL.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. https://en.wikipedia.org/wiki/MySQL.
12. “MongoDB.” Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. https://en.wikipedia.org/wiki/MongoDB.

C++’s history and significant milestones are well documented:

- Wikipedia’s C++ page provides a detailed history of C++ with many citations:

```
https://en.wikipedia.org/wiki/C%2B%2B
```

- Bjarne Stroustrup, C++’s creator, provides a thorough history of the language and its design from inception through C++20:

```
https://www.stroustrup.com/C++.html#design
```

- cppreference.com provides a list of C++ milestones since its inception with many citations:

```
https://en.cppreference.com/w/cpp/language/history
```

1.2 Test-Driving a C++20 Application

In this section, you’ll compile, run and interact with your first C++ application[13]—a guess-the-number game, which picks a random number from 1 to 1,000 and prompts you to guess it. If you guess correctly, the game ends. If you guess incorrectly, the application indicates whether your guess is higher or lower than the correct number. There’s no limit on the number of guesses you can make.

13. We intentionally do not cover the code for this C++ program here. Its purpose is simply to demonstrate compiling and running a program using each compiler we discuss in this section. We present random-number generation in Chapter 5.

Summary of the Compiler and IDE Test-Drives

We’ll show how to compile and execute C++ code using:

- Microsoft Visual Studio 2022 Community edition for Windows (Section 1.2.1),
- Clang in Apple Xcode on macOS (Section 1.2.2),
- GNU `g++` in a shell on Linux (Section 1.2.3),
- `g++` in a shell running inside the GNU Compiler Collection (GCC) Docker container (Section 1.2.4), and
- `clang++` (the command-line version of the Clang C++ compiler) in a shell running inside a Docker container (Section 1.2.5).

You can read only the section that corresponds to your platform. To use the Docker containers for `g++` and `clang++`, you must have Docker installed and running, as discussed in the **Before You Begin** section after the **Preface**.

1.2.1 Compiling and Running a C++20 Application with Visual Studio 2022 Community Edition on Windows

In this section, you'll run a C++ program on Windows using Microsoft Visual Studio 2022 Community edition.[14] There are several versions of Visual Studio available—on some versions, the options, menus and instructions we present might differ slightly. From this point forward, we'll simply say "Visual Studio" or "the IDE."

14. At the time of this writing, the Visual Studio 2022 Community version number was 17.0.5.

Step 1: Checking Your Setup

If you have not already done so, read the **Before You Begin** section to install the IDE and download the book's code examples.

Step 2: Launching Visual Studio

Open Visual Studio from the **Start** menu. Dismiss this initial Visual Studio window by pressing the *Esc* key. Do not click the **X** in the upper-right corner—that will terminate Visual Studio. You can access this window at any time by selecting **File > Start Window**. We use **>** to indicate selecting a menu item from a menu, so **File > Open** means "select the **Open** menu item from the **File** menu."

Step 3: Creating a Project

SE A **project** is a group of related files, such as the C++ source-code files that compose an application. Visual Studio organizes applications into projects and **solutions**. A solution contains one or more projects. Multi-project solutions are used to create large-scale applications. Each application in this book requires only a single-project solution. For our code examples, you'll begin with an **Empty Project** and add files to it. To create a project:

1. Select **File > New > Project...** to display the **Create a New Project** dialog.
2. Select the **Empty Project** template with the tags **C++**, **Windows** and **Console**. This project template is for programs that execute at the command line in a Command Prompt window. Depending on your Visual Studio version and its installed options, many other project templates may be installed. You can filter your choices using the **Search for templates** textbox and the drop-down lists below it. Click **Next** to display the **Configure your new project** dialog.
3. Provide a **Project name** and **Location**. For the **Project name**, we specified `cpp20_test`. For the **Location**, we selected this book's `examples` folder. Click **Create** to open your new project in Visual Studio.

At this point, the Visual Studio creates your project, places its folder in

```
C:\Users\YourUserAccount\Documents\examples
```

(or the folder you specified) and opens the main window.

When you edit C++ code, Visual Studio displays each file as a separate tab within the window. The **Solution Explorer**—docked to Visual Studio's left or right side—is for viewing and managing your application's files. In this book's examples, you'll typically place each program's code files in the **Source Files** folder. If the **Solution Explorer** is not displayed, you can display it by selecting **View > Solution Explorer**.

Step 4: Adding the `GuessNumber.cpp` File to the Project

Next, you'll add `GuessNumber.cpp` to the project you created in *Step 3*. In the **Solution Explorer**:

1. Right-click the **Source Files** folder and select **Add > Existing Item...**.
2. In the dialog that appears, navigate to the `ch01` subfolder of the book's `examples` folder, select `GuessNumber.cpp` and click **Add**.[15]

15. For the multiple source-code-file programs that you'll see in later chapters, select all the files for a given program. When you begin creating programs yourself, you can right-click the **Source Files** folder and select **Add > New Item...** to display a dialog for adding a new file.

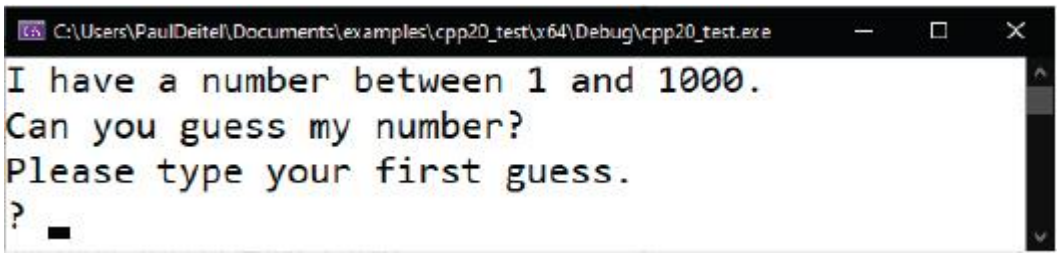
Step 5: Configuring Your Project to Use C++20

The Visual C++ compiler in Visual Studio supports several versions of the C++ standard. For this book, we use C++20, which we must configure in our project's settings:

1. Right-click the project's node— —in the **Solution Explorer** and select **Properties** to display the project's **cpp20_test Property Pages** dialog.
2. In the **Configuration** drop-down list, select **All Configurations**. In the **Platform** drop-down list, select **All Platforms**.
3. In the left column, expand the **C/C++** node, then select **Language**.
4. 20 In the right column, click in the field to the right of **C++ Language Standard**, click the down arrow, then select **ISO C++20 Standard (/std:c++20)** and click **OK**.

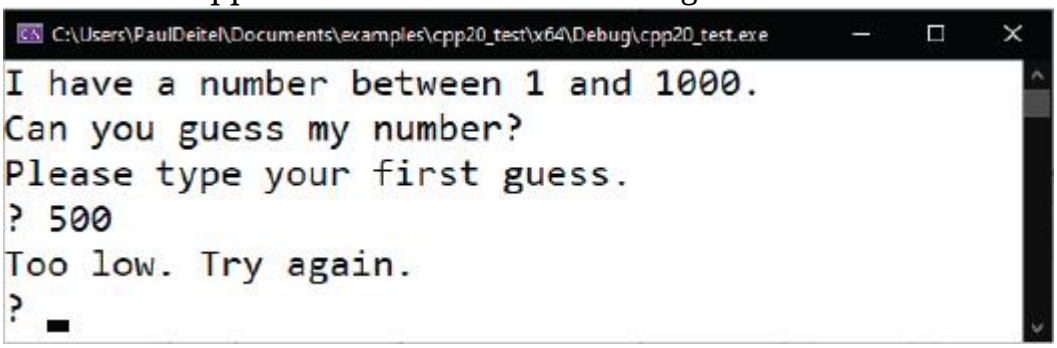
Step 6: Compiling and Running the Project

To compile and run the project so you can test-drive the application, select **Debug > Start without debugging** or type *Ctrl* + *F5*. If the program compiles correctly, Visual Studio opens a Command Prompt window and executes the program. We changed the Command Prompt's color scheme and font size for readability:


C:\Users\PaulDeitel\Documents\examples\cpp20_test\x64\Debug\cpp20_test.exe
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?

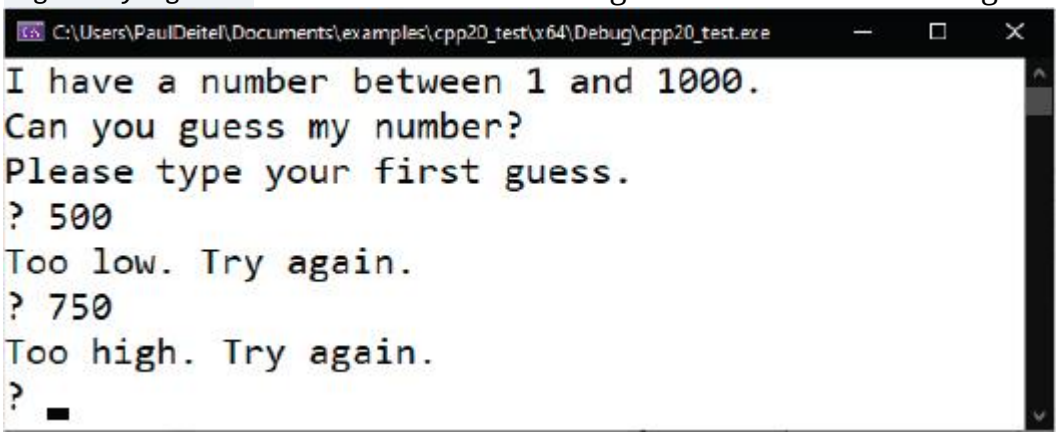
Step 7: Entering Your First Guess

At the `?` prompt, type **`500`** and press *Enter*—the outputs will vary each time you run the program. In our case, the application displayed `"Too low. Try again."` to indicate the value was less than the number the application chose as the correct guess:



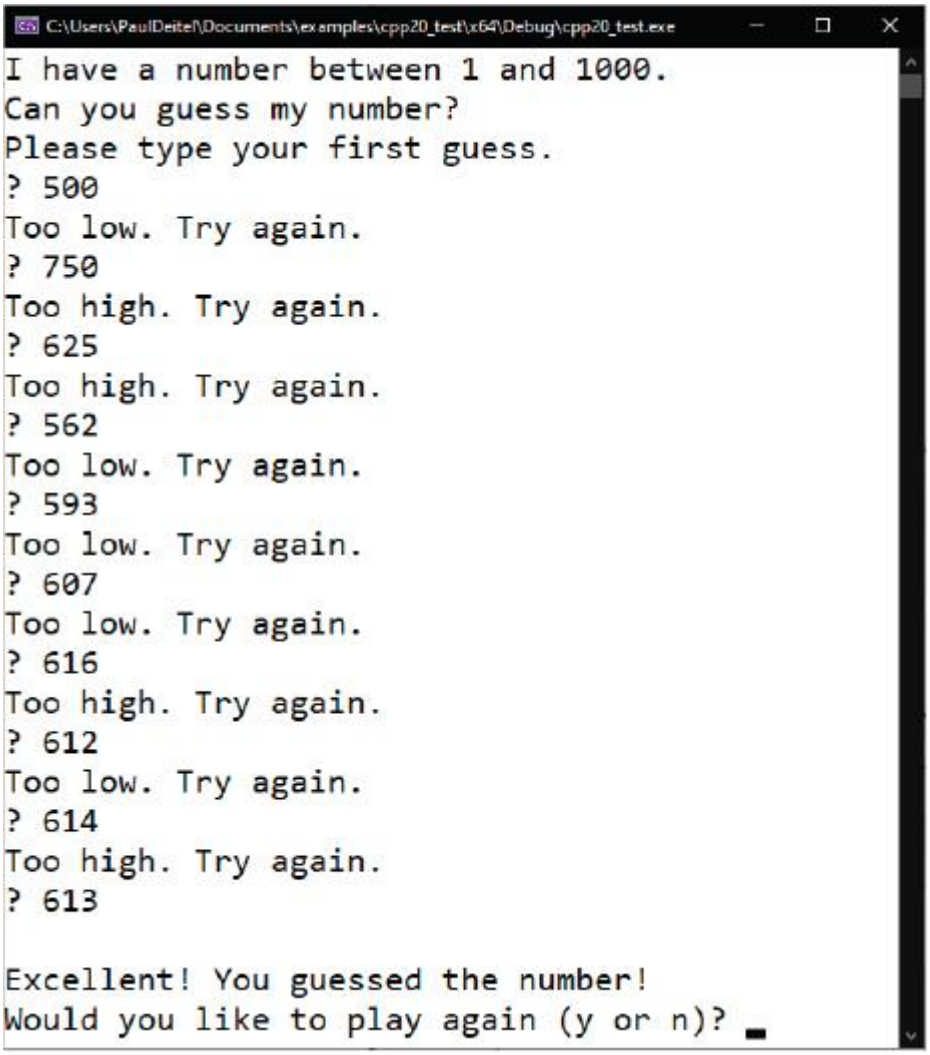
Step 8: Entering Another Guess

At the next prompt, if your system said the first guess was too low, type **`750`** and press *Enter*; otherwise, type **`250`** and press *Enter*. In our case, we entered **`750`**, and the application displayed `"Too high. Try again."` because the value was greater than the correct guess:



Step 9: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays `"Excellent! You guessed the number!"`:



Step 10: Playing the Game Again or Exiting the Application

After guessing the correct number, the application asks if you'd like to play another game. At the `"Would you like to play again (y or n)?"` prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application.

Reusing This Project for Subsequent Examples

You can follow the steps in this section to create a separate project for every application in the book. However, you may find it more convenient for our examples to remove the current program from the project, then add a new program. To remove a file from your project (but not your system), select it in the **Solution Explorer**, then press *Del* (or *Delete*). You can then repeat *Step 4* to add a different program to the project.

Using Ubuntu Linux in the Windows Subsystem for Linux

Some Windows users may want to use the GNU `gcc` compiler on Windows. You can do this using the **GNU Compiler Collection Docker container** (Section 1.2.4), or you can use `gcc` in Ubuntu Linux running in the **Windows Subsystem for Linux**. To install the Windows Subsystem for Linux, follow the instructions at

```
https://docs.microsoft.com/en-us/windows/wsl/install
```

Once you install and launch the **Ubuntu** app on your Windows System, you can use the following command to change to the folder containing the test-drive code example on your Windows system:

```
cd /mnt/c/Users/YourUseName/Documents/examples/ch01
```

Then you can continue with *Step 2* in Section 1.2.3.

1.2.2 Compiling and Running a C++20 Application with Xcode on macOS

In this section, you'll run a C++ program on macOS using Apple's version of the Clang compiler in the Apple Xcode IDE.[16]

16. At the time of this writing, the Xcode version was 13.2.1.

Step 1: Checking Your Setup

If you have not already done so, read the **Before You Begin** section to install the IDE and download the book's code examples.

Step 2: Launching Xcode

Open a Finder window, select **Applications** and double-click the Xcode icon:

If this is your first time running Xcode, the **Welcome to Xcode** window appears. Close this window—you can access it by selecting **Window > Welcome to Xcode**. We use the **>** character to indicate selecting a menu item from a menu. For example, **File > Open...** indicates that you should select the **Open...** menu item from the **File** menu.

Step 3: Creating a Project

A **project** is a group of related files, such as the C++ source-code files that compose an application. The Xcode projects we created for this book's examples are **Command Line Tool** projects that you'll execute directly in the IDE. To create a project:

1. Select **File > New > Project...**.
2. At the top of the **Choose a template for your new project** dialog, click **macOS**.
3. Under **Application**, click **Command Line Tool** and click **Next**.
4. For **Product Name**, enter a name for your project—we specified `cpp20_test`.
5. In the **Language** drop-down list, select **C++**, then click **Next**.
6. Specify where you want to save your project. We selected the `examples` folder containing this book's code examples.
7. Click **Create**.

Xcode creates your project and displays the **workspace window** initially showing three areas—the **Navigator area** (left), **Editor area** (middle) and **Utilities area** (right).

The left-side **Navigator** area has icons at its top for the navigators that can be displayed there. For this book, you'll primarily work with two of these navigators:

- **Project** ( )—Shows all the files and folders in your project.
- **Issue** ( )—Shows you warnings and errors generated by the compiler.

Clicking a navigator button displays the corresponding navigator panel.

The middle **Editor** area is for managing project settings and editing source code. This area is always displayed in your workspace window. When you select a file in the **Project** navigator, the file's contents display in the **Editor** area. The right-side **Utilities** area typically displays **inspectors**. For example, if you were building an iPhone app that contained a touchable button, you'd be able to configure the button's properties (its label, size, position, etc.) in this area. You will not use the **Utilities** area in this book. There's also a **Debug area** where you'll interact with the running guess-the-number program. This will appear below the **Editor** area.

The workspace window's toolbar contains options for executing a program, displaying the progress of tasks executing in Xcode, and hiding or showing the left (Navigator) and right (Utilities) areas.

Step 4: Configuring the Project to Compile Using C++20

20 The Apple Clang compiler in Xcode supports several versions of the C++ standard. For this book, we use C++20, which we must configure in our project's settings:

1. In the **Project** navigator, select your project's name (`cpp20_test`).
2. In the **Editors** area's left side, select your project's name under **TARGETS**.
3. At the top of the **Editors** area, click **Build Settings**, and just below it, click **All**.
4. Scroll to the **Apple Clang - Language - C++** section.
5. Click the value to the right of **C++ Language Dialect** and select **GNU++20 [-std=gnu++20]**.
6. Click the value to the right of **C++ Standard Library** and select **Compiler Default**.

Step 5: Deleting the `main.cpp` File from the Project

By default, Xcode creates a `main.cpp` source-code file containing a simple program that displays `"Hello, World!"`. You won't use `main.cpp` in this test-drive, so you should delete the file. In the **Project** navigator, right-click the `main.cpp` file and select **Delete**. In the dialog that appears, select **Move to Trash**. The file will not be removed from your system until you empty your trash.

Step 6: Adding the `GuessNumber.cpp` File into the Project

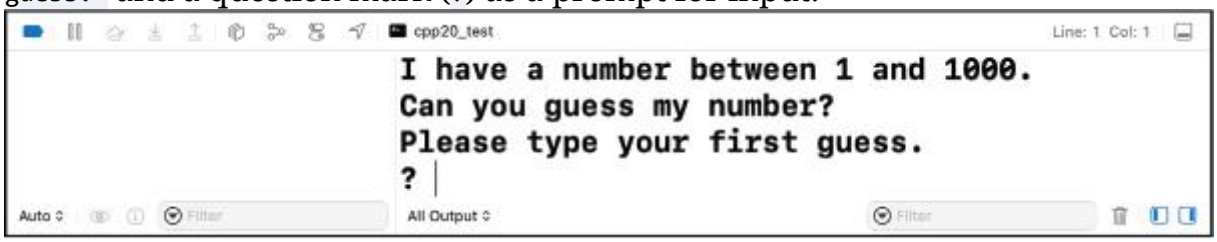
In a Finder window, open the `ch01` folder in the book's `examples` folder, then drag `GuessNumber.cpp` onto the **cpp20_test** folder in the **Project** navigator. In the dialog that appears, ensure that **Copy items if needed** is checked, then click **Finish**.[17]

17. For the multiple source-code-file programs that you'll see later in the book, drag all the files for a given program to the project's folder. When you begin creating your own programs, you can right-click the project's folder and select **New File...** to display a dialog for adding a new file.

Step 7: Compiling and Running the Project

To compile and run the project so you can test-drive the application, simply click the run ( ) button on Xcode's toolbar. If the program compiles correctly, Xcode opens the **Debug** area and executes the

program in the right half of the **Debug** area, and the application displays `"Please type your first guess."` and a question mark (`?`) as a prompt for input:



Step 8: Entering Your First Guess

Click the **Debug** area, then type **`500`** and press *Return*—the outputs will vary each time you run the program. In our case, the application displayed `"Too high. Try again."` because the value was more than the number the application chose as the correct guess.

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Step 9: Entering Another Guess

At the next prompt, if your system said the first guess was too low, type **`750`** and press *Enter*; otherwise, type **`250`** and press *Enter*. In our case, we entered **`250`**, and the application displayed `"Too high. Try again."` because the value was greater than the correct guess:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

Step 10: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays `"Excellent! You guessed the number."`:

```
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
? 125
Too low. Try again.
? 187
Too high. Try again.
? 156
Too high. Try again.
? 140
Too low. Try again.
? 148
Too high. Try again.
? 144
Too high. Try again.
? 142
Too low. Try again.
? 143

Excellent! You guessed the number!
Would you like to play again (y or n)?
```

Playing the Game Again or Exiting the Application

After guessing the correct number, the application asks if you'd like to play another game. At the `"Would you like to play again (y or n)?"` prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application.

Reusing This Project for Subsequent Examples

You can follow the steps in this section to create a separate project for every application in the book. However, for our examples, you may find it more convenient to remove the current program from the project, then add a new one. To remove a file from your project (but not your system), right-click the file in the **Project** navigator and select **Delete**. In the dialog that appears, select **Remove Reference**. You can then repeat *Step 6* to add a different program to the project.

1.2.3 Compiling and Running a C++20 Application with GNU C++ on Linux

In this section, you'll run a C++ program in a Linux shell using the GNU C++ compiler (`g++`).[18] For this test-drive, we assume that you've read the **Before You Begin** section and that you've placed the book's examples in your user account's `Documents/examples` folder.

18. At the time of this writing, the current `g++` version was 11.2. You can determine your system's g++ version number with the command `g++ --version`. If you have an older version of g++, consider searching online for the instructions to upgrade the GNU Compiler Collection (GCC) for your Linux distribution or consider using the GCC Docker container discussed in Section 1.2.4.

Step 1: Changing to the `ch01` Folder

From a Linux shell, use the `cd` command to change to the `ch01` subfolder of the book's `examples` folder:

```
~$ cd ~/Documents/examples/ch01
~/Documents/examples/ch01$
```

In this section's figures, we use **bold** to highlight the user inputs. The prompt in our Ubuntu Linux shell uses a tilde (~) to represent the home directory. Each prompt ends with the dollar sign ($). The prompt may differ on your Linux system.

Step 2: Compiling the Application

Before running the application, you must first compile it with the `g++` command:[19]

19. If you have multiple `g++` versions installed, you might need to use `g++-##`, where `##` is the `g++` version number. For example, the command `g++-11` might be required to run the latest version of `g++` 11.x on your computer.

- The `-std=c++20` option indicates that we're using C++20.
- The `-o` option names the executable file (`GuessNumber`) that you'll use to run the program. If you do not include this option, `g++` automatically names the executable `a.out`.

```
~/Documents/examples/ch01$ g++ -std=c++20 GuessNumber.cpp -o GuessNumber
~/Documents/examples/ch01$
```

Step 3: Running the Application

Type `./GuessNumber` at the prompt and press *Enter* to run the program:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

The `./` before `GuessNumber` tells Linux to run `GuessNumber` from the current directory.

Step 4: Entering Your First Guess

The application displays `"Please type your first guess."`, then displays a question mark (`?`) as a prompt on the next line. At the prompt, enter **500**—the outputs will vary each time you run the program:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

In our case, the application displayed `"Too high. Try again."` because the value entered was greater than the number the application chose as the correct guess.

Step 5: Entering Another Guess

At the next prompt, if your system said the first guess was too low, type **750** and press *Enter*; otherwise, type **250** and press *Enter*. In our case, we entered **250**, and the application displayed `"Too high. Try again."` because the value was greater than the correct guess:

```
~/Documents/examples/ch01$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

Step 6: Entering Additional Guesses

Continue to play the game by entering values until you guess the correct number. When you guess correctly, the application displays `"Excellent! You guessed the number."`:

```
? 125
Too high. Try again.
? 62
Too low. Try again.
? 93
Too low. Try again.
? 109
Too high. Try again.
? 101
Too low. Try again.
? 105
Too high. Try again.
? 103
Too high. Try again.
? 102
Excellent! You guessed the number!
Would you like to play again (y or n)?
```

Step 7: Playing the Game Again or Exiting the Application

After guessing the correct number, the application asks if you'd like to play another game. At the `"Would you like to play again (y or n)?"` prompt, entering **y** causes the application to choose a new number and start a new game. Entering **n** terminates the application and returns you to the shell.

1.2.4 Compiling and Running a C++20 Application with g++ in the GCC Docker Container

You can use the latest GNU C++ compiler on your system, regardless of your operating system. One of the most convenient cross-platform ways to do this is by using the **GNU Compiler Collection (GCC) Docker container**. This section assumes you've already installed **Docker Desktop** (Windows or macOS) or **Docker Engine** (Linux), as discussed in the **Before You Begin** section that follows the **Preface**.

Executing the GNU Compiler Collection (GCC) Docker Container

Open a **Command Prompt** (Windows), **Terminal** (macOS/Linux) or **shell** (Linux), then perform the following steps to launch the GCC Docker container:

1. Use `cd` to navigate to the `examples` folder containing this book's examples.
2. Windows users: Launch the GCC Docker container with the command[20]

```
docker run --rm -it -v "%CD%":/usr/src gcc:latest
```

20. A notification might appear asking you to allow Docker to access the files in the current folder. You must allow this; otherwise, you will not be able to access our source-code files in Docker.

3. macOS/Linux users: Launch the GCC Docker container with the command

```
docker run --rm -it -v "$(pwd)":/usr/src gcc:latest
```

In the preceding commands:

- `--rm` cleans up the container's resources when you eventually shut it down.
- `-it` runs the container in interactive mode, so you can enter commands to change folders and to compile and run programs using the GNU C++ compiler.
- `-v "%CD%":/usr/src` (Windows) or `-v "$(pwd)":/usr/src` (macOS/Linux) allows the Docker container to access the files in the folder from which you executed the `docker run` command. In the Docker container, you'll navigate with the `cd` command to subfolders of `/usr/src` to compile and run the book's examples. In other words, your local system folder will be mapped to the `/usr/src` folder in the Docker container.
- `gcc:latest` is the container name. The `:latest` specifies that you want to use the most up-to-date version of the `gcc` container.[21]

21. If you'd like to keep your GCC container up-to-date with the latest release, you can execute the command `docker pull gcc:latest` before running the container.

Once the container is running, you'll see a prompt similar to:

```
root@67773f59d9ea:/#
```

The container uses a Linux operating system. Its prompt displays the current folder location between the `:` and `#`.

Changing to the `ch01` Folder in the Docker Container

The `docker run` command specified above attaches your `examples` folder to the container's `/usr/src` folder. In the Docker container, use the `cd` command to change to the `ch01` subfolder of `/usr/src`:

```
root@01b4d47cadc6:/# cd /usr/src/ch01
root@01b4d47cadc6:/usr/src/ch01#
```

To compile, run and interact with the `GuessNumber` application in the Docker container, follow *Steps 2–7* of Section 1.2.3's GNU C++ Test-Drive.

Terminating the Docker Container

You can terminate the Docker container by typing *Ctrl + d* at the container's prompt.

1.2.5 Compiling and Running a C++20 Application with `clang++` in a Docker Container

As with `g++`, you can use the latest LLVM/Clang C++ (`clang++`) command-line compiler on your system, regardless of your operating system. Currently, the LLVM/Clang team does not have an official Docker container, but many working containers are available on https://hub.docker.com. This section assumes you've already installed **Docker Desktop** (Windows or macOS) or **Docker Engine** (Linux), as discussed in the **Before You Begin** section that follows the **Preface**.

We used the most recent and widely downloaded one containing `clang++` version 13, which you can get via the following command:[22]

22. The version of the Clang C++ compiler used in Xcode is not the most up-to-date version, so it does not have as many C++20 features implemented as the version directly from the LLVM/Clang team. Also, at the time of this writing, using `"latest"` rather than `"13"` in the `docker pull` command gives you a Docker container with `clang++` 12, not 13.

```
docker pull teeks99/clang-ubuntu:13
```

Executing the `teeks99/clang-ubuntu` Docker Container

Open a **Command Prompt** (Windows), **Terminal** (macOS/Linux) or **shell** (Linux), then perform the following steps to launch the **`teeks99/clang-ubuntu` Docker container**:

1. Use `cd` to navigate to the `examples` folder containing this book's examples.
2. Windows users: Launch the Docker container with the command[23]

23. A notification will appear asking you to allow Docker to access the files in the current folder. You must allow this; otherwise, you will not be able to access our source-code files in Docker.

```
docker run --rm -it -v "%CD%":/usr/src teeks99/clang-ubuntu:13
```

3. macOS/Linux users: Launch the Docker container with the command

```
docker run --rm -it -v "$(pwd)":/usr/src teeks99/clang-ubuntu:13
```

In the preceding commands:

- `--rm` cleans up the container's resources when you eventually shut it down.
- `-it` runs the container in interactive mode, so you can enter commands to change folders and to compile and run programs using the `clang++` compiler.
- `-v "%CD%":/usr/src` (Windows) or `-v "$(pwd)":/usr/src` (macOS/Linux) allows the Docker container to access the files in the folder from which you executed the `docker run` command. In the Docker container, you'll navigate with the `cd` command to subfolders of `/usr/src` to compile and run the book's examples. In other words, your local system folder will be mapped to the `/usr/src` folder in the Docker container.
- `teeks99/clang-ubuntu:13` is the container name.

Once the container is running, you'll see a prompt similar to:

```
root@9753bace2e87:/#
```

The container uses a Linux operating system. Its prompt displays the current folder location between the `:` and `#`.

Changing to the `ch01` Folder in the Docker Container

The `docker run` command specified above attaches your `examples` folder to the container's `/usr/src` folder. In the Docker container, use the `cd` command to change to the `ch01` subfolder of `/usr/src`:

```
root@9753bace2e87:/# cd /usr/src/ch01
root@9753bace2e87:/usr/src/ch01#
```

Compiling the Application

Before running the application, you must first compile it. This container uses the command `clang++-13`, as in

```
clang++-13 -std=c++20 GuessNumber.cpp -o GuessNumber
```

where:

- The `-std=c++20` option indicates that we're using C++20.
- The `-o` option names the executable file (`GuessNumber`) that you'll use to run the program. If you do not include this option, `clang++` automatically names the executable `a.out`.

Running the Application

To run and interact with the `GuessNumber` application in the Docker container, follow *Steps 3–7* of Section 1.2.3's GNU C++ Test-Drive.

Terminating the Docker Container

You can terminate the Docker container by typing *Ctrl + d* at the container's prompt.

1.3 Moore's Law, Multi-Core Processors and Concurrent Programming

Perf Many of today's personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime. *Supercomputers* are already performing *thousands of trillions (quadrillions)* of instructions per second. The Japanese Fugaku supercomputer can perform over 442 quadrillion calculations per second (442.01 *petaflops*).[24] To put that in perspective, **the Fugaku supercomputer can perform in one second about 40 million calculations for every person on the planet!** And supercomputing "upper limits" are growing quickly.

24. "Top500." Wikipedia. Wikimedia Foundation. Accessed January 10, 2022. https://en.wikipedia.org/wiki/TOP500.

Moore's Law

Every year, you probably expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the hardware supporting these technologies. Over the years, hardware costs have fallen rapidly.

Perf For decades, computer processing power approximately doubled inexpensively every couple of years. This remarkable trend often is called **Moore's law**, named for Gordon Moore, co-founder of Intel and the person who identified the trend in the 1960s. Intel is a leading manufacturer of processors in today's computers and embedded systems, such as smart home appliances, home security systems, robots, intelligent traffic intersections and more. **Moore's law and related observations** apply especially to

- the amount of memory that computers have for programs and data,
- the amount of secondary storage they have to hold programs and data, and
- their processor speeds—that is, the speeds at which computers execute programs to do their work.

Perf Key executives at computer-processor companies NVIDIA and Arm have indicated that Moore's law no longer applies.[25,26] Computer processing power continues to increase but now relies on new processor designs, such as multi-core processors.

25. "Moore's Law Turns 55: Is It Still Relevant?" Accessed November 2, 2020. https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant.
26. "Moore's Law Is Dead: Three Predictions About the Computers of Tomorrow." Accessed November 2, 2020. https://www.techrepublic.com/article/moores-law-is-dead-three-predictions-about-the-computers-of-tomorrow/.

Multi-Core Processors and Performance

Most computers today have **multi-core processors** that economically implement multiple processors on a single integrated circuit chip. A dual-core processor has two CPUs, a quad-core processor has four, and an octa-core processor has eight. Our primary testing computer uses an eight-core Intel processor. Apple's recent M1 Pro and M1 Max processors have 10-core CPUs. In addition, the top-of-the-line M1 Pro has a 16-core GPU, while the top-of-the-line M1 Max processor has a 32-core GPU, and both have a 16-core "neural engine" for machine learning.[27,28] Intel has some processors with up to 72 cores[29] and is working on processors with up to 80.[30] AMD is working on processors with 192 and 256 cores.[31] The number of cores will continue to grow.

27. Juli Clover, "Apple's M1 Pro Chip: Everything You Need to Know," November 3, 2021. Accessed January 19, 2022. https://www.macrumors.com/guide/m1-pro/.
28. "Apple M1 Pro and M1 Max." Wikipedia. Wikimedia Foundation. Accessed January 19, 2022. https://en.wikipedia.org/wiki/Apple_M1_Pro_and_M1_Max.
29. "Intel® Xeon Phi™ Processors." Accessed November 28,2021. https://ark.intel.com/content/www/us/en/ark/products/series/132784/intel-xeon-phi-72x5-processor-family.html.
30. Anton Shilov, "Intel's Sapphire Rapids Could Have 72–80 Cores, According to New Die Shots," April 30, 2021. Accessed November 28, 2021. https://www.tomshardware.com/news/intel-sapphire-rapids-could-feature-80-cores.
31. Joel Hruska, "Future 256-Core AMD Epyc CPU Might Sport Remarkably Low 600W TDP," November 1, 2021. Accessed November 28, 2021. https://www.extremetech.com/computing/328692-future-256-core-amd-epyc-cpu-might-sport-remarkably-low-600w-tdp.

Perf In multi-core systems, the hardware can put multiple processors to work truly simultaneously on different parts of your task, thereby enabling your program to complete faster. **To take full advantage of multi-core architecture, you need to write multithreaded applications.** When a

program splits tasks into separate threads, a multi-core system can run those threads in parallel when a sufficient number of cores is available.

11 SE Interest in multithreading is rising quickly because of the proliferation of multi-core systems. Standard C++ multithreading was one of the most significant updates introduced in C++11. Each subsequent C++ standard has added higher-level capabilities to simplify multithreaded application development. Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, discusses creating and managing multithreaded C++ applications. Chapter 18 introduces C++20 coroutines, which enable concurrent programming with a simple sequential-like coding style.

1.4 A Brief Refresher on Object Orientation

Building software quickly, correctly and economically remains an elusive goal at a time when demands for new and more powerful software are soaring. **Objects**, or more precisely—as we'll see in Chapter 9—the **classes** objects come from, are essentially **reusable** software components. There are date objects, time objects, audio objects, video objects, automobile objects, people objects, etc. Almost any **noun** can be reasonably represented as a software object in terms of **attributes** (e.g., name, color and size) and **behaviors** (e.g., calculating, moving and communicating). Software developers have discovered that using a modular, object-oriented design-and-implementation approach can make software development groups much more productive than was possible with earlier techniques—object-oriented programs are often easier to understand, correct and modify.

The Automobile as an Object

Let's begin with a simple analogy. Suppose you want to drive a car and make it go faster by pressing its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to **design** it. A car typically begins as engineering drawings, similar to the **blueprints** that describe the design of a house. These drawings include the design for an accelerator pedal. The pedal **hides** from the driver the complex mechanisms that make the car go faster, just as the brake pedal hides the mechanisms that slow the car, and the steering wheel hides the mechanisms that turn the car. This enables people with little or no knowledge of how engines, braking and steering mechanisms work to drive a car easily.

Before you can drive a car, it must be **built** from the engineering drawings that describe it. A completed car has an **actual** accelerator pedal to make the car go faster, but even that's not enough—the car won't accelerate on its own (hopefully!), so the driver must **press** the pedal to accelerate the car.

Functions, Member Functions and Classes

Let's use our car example to introduce some key object-oriented programming concepts. Performing a task in a program requires a function. The function houses the program statements that perform its task. It **hides** these statements from its user, just as the accelerator pedal of a car hides from the driver the mechanisms of making the car go faster. In C++, we often create a program unit called a class to house the set of functions that perform the class's tasks—these are known as the class's member functions. For example, a class representing a bank account might contain a member function to **deposit** money to an account, another to **withdraw** money from an account and a third to **query** the account's current balance. A class is similar to a car's engineering drawings, which house the design of an accelerator pedal, brake pedal, steering wheel, and so on.

Instantiation

Just as someone has to **build a car** from its engineering drawings before you can drive a car, you must **build an object** from a class before a program can perform the tasks that the class's member functions define. The process of doing this is called **instantiation**. An object is then referred to as an instance of its class.

Reuse

SE Just as a car's engineering drawings can be **reused** many times to build many cars, you can **reuse** a class many times to build many objects. Reuse of existing classes when building new classes and programs saves time and effort. Reuse also helps you build more reliable and effective systems because existing classes and components often have been extensively **tested**, **debugged** and **performance tuned**. Just as the notion of **interchangeable parts** was crucial to the Industrial Revolution, reusable classes are crucial to the software revolution that has been spurred by object technology.

Messages and Member-Function Calls

When you drive a car, pressing its gas pedal sends a **message** to the car to perform a task—that is, to "go faster." Similarly, you **send messages to an object**. Each message is implemented as a member-function call that tells a member function of the object to perform its task. For example, a program might call a particular bank-account object's **deposit** member function to increase the account's balance by the deposit amount.

Attributes and Data Members

Besides having capabilities to accomplish tasks, a car also has **attributes**, such as its color, number of doors, amount of gas in its tank, current speed and record of total miles driven (i.e., its odometer reading). Like its capabilities, the car's attributes are represented as part of its design in its engineering diagrams (which, for example, include an odometer and a fuel gauge). As you drive a car, these attributes are "carried along" with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars.

An object, similarly, has attributes that it carries along as it's used in a program. These attributes are specified as part of the object's class. For example, a bank-account object has a **balance attribute** representing the amount of money in the account. Each bank-account object knows the balance in the account it represents, but not the balances of the other accounts in the bank. Attributes are specified by the class's data members.

Encapsulation

SE Classes encapsulate (i.e., wrap) attributes and member functions into objects created from those classes—an object's attributes and member functions are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented internally. Those details are **hidden** within the objects themselves. This information hiding, as we'll see, is crucial to good software engineering.

Inheritance

A new class of objects can be created quickly and conveniently by inheritance. The new class absorbs the characteristics of an existing class, possibly customizing them and adding unique characteristics

of its own. In our car analogy, an object of class "convertible" certainly **is an** object of the more **general** class "automobile," but more **specifically**, the roof can be raised or lowered.

Object-Oriented Analysis and Design

Soon you'll be writing programs in C++. How will you create the **code** for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the book's early chapters), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of thousands of software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

SE To create the best solutions, you should follow a detailed **analysis** process for determining your project's **requirements** (i.e., defining **what** the system is supposed to do) and developing a **design** that satisfies them (i.e., deciding **how** the system should do it). Ideally, you'd go through this process and carefully review the design (and have your design reviewed by other software professionals) before writing any code. If this process involves analyzing and designing your system from an object-oriented point of view, it's called an **object-oriented analysis and design (OOAD) process**. Languages like C++ are object-oriented. Programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

1.5 Wrap-Up

In this introductory chapter, you saw how to compile and run applications using our three preferred compilers—Visual C++ in Visual Studio 2022 Community edition on Windows, Clang in Xcode on macOS and GNU `g++` on Linux. We pointed you to a Microsoft resource for installing Ubuntu Linux using the Windows Subsystem for Linux so you can run `g++` on Windows. We also demonstrated how to launch cross-platform Docker containers so you can use the latest `g++` and `clang++` versions on Windows, macOS or Linux.

We pointed you to several resources for learning about C++'s history and design, including those provided by Bjarne Stroustrup, C++'s creator. Next, we discussed Moore's law, multi-core processors and why Modern C++'s concurrent programming features are crucial for taking advantage of the power multi-core processors provide. Finally, we provided a brief refresher on object-oriented programming concepts and terminology we'll use throughout the book.

In the next chapter, we introduce C++ programming with basic input and output statements, fundamental data types, arithmetic, decision making and our first "Objects Natural" case study on using objects of C++ standard library class `string`.

2. Intro to C++20 Programming

Objectives

In this chapter, you'll:

- Write simple C++ applications.
- Use input and output statements.
- Use fundamental data types.
- Use arithmetic operators.
- Understand the precedence of arithmetic operators.
- Write decision-making statements.
- Use relational and equality operators.
- Begin appreciating the "Objects Natural" learning approach by creating and using objects of the C++ standard library's `string` class before creating your own custom classes.

Outline

2.1 Introduction
2.2 First Program in C++: Displaying a Line of Text
2.3 Modifying Our First C++ Program
2.4 Another C++ Program: Adding Integers
2.5 Arithmetic
2.6 Decision Making: Equality and Relational Operators
2.7 Objects Natural: Creating and Using Objects of Standard-Library Class `string`
2.8 Wrap-Up

2.1 Introduction

This chapter presents several code examples that demonstrate how your programs can display messages and obtain data from the user for processing. The first three examples display messages on the screen. The next obtains two numbers from a user at the keyboard, calculates their sum and displays the result—the accompanying discussion introduces C++'s arithmetic operators. The fifth example demonstrates decision making by showing you how to compare two numbers, then display messages based on the comparison results.

The "Objects Natural" Learning Approach

In your programs, you'll create and use many objects of preexisting carefully-developed-and-tested classes that enable you to perform significant tasks with minimal code. These classes typically come from:

- the C++ standard library,
- platform-specific libraries (such as those provided by Microsoft for creating Windows applications or by Apple for creating macOS applications), and

- free third-party libraries often created by the massive open-source communities that have developed around all major contemporary programming languages.

To help you appreciate this style of programming early in the book, you'll create and use objects of preexisting C++ standard library classes before creating your own custom classes. We call this the "Objects Natural" approach. You'll begin by creating and using `string` objects in this chapter's final example. In later chapters, you'll create your own custom classes. You'll see that C++ enables you to craft valuable classes for your own use and for reuse by other programmers.

Compiling and Running Programs

For instructions on compiling and running programs in Microsoft Visual Studio, Apple Xcode and GNU C++, see the test-drives in Chapter 1 or our video instructions at:

```
http://deitel.com/c-plus-plus-20-for-programmers
```

2.2 First Program in C++: Displaying a Line of Text

Consider a simple program that displays a line of text (Fig. 2.1). The line numbers are not part of the program.

```
1  // fig02_01.cpp
2  // Text-printing program.
3  #include <iostream> // enables program to output data to the screen
4
5  // function main begins program execution
6  int main() {
7     std::cout << "Welcome to C++!\n"; // display message
8
9     return 0; // indicate that program ended successfully
10 } // end function main
```

```
Welcome to C++!
```

Fig. 2.1 Text-printing program.

Comments

Lines 1 and 2

```
// fig02_01.cpp
// Text-printing program.
```

both begin with `//`, indicating that the remainder of each line is a **comment**. In each of our programs, the first-line comment contains the program's file name. The comment `"Text-printing program."` describes the program's purpose. A comment beginning with `//` is called a **single-line**

comment because it terminates at the end of the current line. You can create single or **multiline comments** by enclosing them in `/*` and `*/`, as in

```
/* fig02_01.cpp: Text-printing program. */
```

or

```
/* fig02_01.cpp
   Text-printing program. */
```

`#include` Preprocessing Directive

Line 3

```
#include <iostream> // enables program to output data to the screen
```

is a **preprocessing directive**—that is, a message to the C++ preprocessor, which the compiler invokes before compiling the program. This line notifies the preprocessor to include in the program the contents of the **input/output stream header** `<iostream>`. This header is a file containing information the compiler requires when compiling any program that outputs data to the screen or inputs data from the keyboard using C++'s stream input/output. The program in Fig. 2.1 outputs data to the screen. Chapter 5 discusses headers in more detail, and online Chapter 19 explains the contents of `<iostream>` in more detail.

Blank Lines and Whitespace

Line 4 is simply a blank line. You use blank lines, spaces and tabs to make programs easier to read. Together, these characters are known as **whitespace**—they're normally ignored by the compiler.

The `main` Function

Line 6

```
int main() {
```

is a part of every C++ program. The parentheses after `main` indicate that it's a **function**. C++ programs typically consist of one or more functions and classes. Exactly one function in every program must be named `main`, which is where C++ programs begin executing. The keyword `int` indicates that after `main` finishes executing, it "returns" an integer (whole number) value. **Keywords** are reserved by C++ for a specific use. We show the complete list of C++ keywords in Chapter 3. We'll explain what it means for a function to "return a value" when we demonstrate how to create your own functions in Chapter 5. For now, simply include the keyword `int` to the left of `main` in each of your programs.

The **left brace**, `{`, (end of line 6) must *begin* each function's **body**, which contains the instructions the function performs. A corresponding **right brace**, `}`, (line 10) must *end* each function's body.

An Output Statement

Line 7

```
std::cout << "Welcome to C++!\n"; // display message
```

displays the characters contained between the double quotation marks. Together, the quotation marks and the characters between them are called a **string**, a **character string** or a **string literal**. We refer to characters between double quotation marks simply as strings. Whitespace characters in strings are not ignored by the compiler.

Err The entire line 7—including `std::cout`, the **`<<` operator**, the string `"Welcome to C++!\n"` and the **semicolon** (`;`)—is called a **statement**. Most C++ statements end with a semicolon. Omitting the semicolon at the end of a C++ statement when one is needed is a syntax error. Preprocessing directives (such as `#include`) are not C++ statements and do not end with a semicolon.

Typically, output and input in C++ are accomplished with **streams** of data. When the preceding statement executes, it sends the stream of characters `Welcome to C++!\n` to the **standard output stream object** (**`std::cout`**), which is normally "connected" to the screen.

Indentation

Indent each function's body one level within the braces that delimit the body. This makes a program's functional structure stand out, making the program easier to read. Set a convention for the size of indent you prefer, then apply it uniformly. The tab key may be used to create indents, but tab stops may vary. We prefer three spaces per level of indent.

The `std` Namespace

The `std::` before `cout` is required when we use names that we've brought into the program from standard-library headers like `<iostream>`. The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to **namespace `std`**.[1] The names `cin` (the standard input stream) and `cerr` (the standard error stream)—introduced in Chapter 1—also belong to namespace `std`. We discuss namespaces in Chapter 16. For now, you should simply remember to include `std::` before each mention of `cout`, `cin` and `cerr` in a program. This can be cumbersome—we'll soon introduce `using` declarations and the `using` directive, which will enable you to omit `std::` before each use of a name in the `std` namespace.

1. We pronounce "`std::`" as "standard," rather as its individual letters `s`, `t` and `d`.

The Stream Insertion Operator and Escape Sequences

In a `cout` statement, the `<<` operator is referred to as the **stream insertion operator**. When this program executes, the value to the operator's right (the right **operand**) is inserted in the output stream. Notice that the `<<` operator points toward where the data goes. A string's characters normally display exactly as typed between the double quotes. However, the characters `\n` are *not* displayed in Fig. 2.1's sample output. The backslash (`\`) is called an **escape character**. It indicates that a "special" character is to be output. When a backslash is encountered in a string, the next character is combined with the backslash to form an **escape sequence**. The escape sequence `\n` means **newline**. It causes

the **cursor** (i.e., the current screen-position indicator) to move to the beginning of the next line on the screen. Some common escape sequences are shown in the following table:

Escape sequence	Description
`\n`	Newline. Positions the screen cursor to the beginning of the next line.
`\t`	Horizontal tab. Moves the screen cursor to the next tab stop.
`\r`	Carriage return. Positions the screen cursor to the beginning of the current line; does not advance to the next line.
`\a`	Alert. Sounds the system bell.
`\\`	Backslash. Includes a backslash character in a string.
`\'`	Single quote. Includes a single-quote character in a string.
`\"`	Double quote. Includes a double-quote character in a string.

The `return` Statement

Line 9

```
return 0; // indicate that program ended successfully
```

is one of several means we'll use to **exit a function**. In this **`return` statement** at the end of `main`, the value `0` indicates that the program terminated successfully. If program execution reaches `main`'s closing brace without encountering a `return` statement, C++ treats that the same as encountering `return 0;` and assumes the program terminated successfully. So, we omit `main`'s `return` statement in subsequent programs that terminate successfully.

2.3 Modifying Our First C++ Program

The next two examples modify the program of Fig. 2.1. The first displays text on one line using multiple statements. The second displays text on several lines using one statement.

Displaying a Single Line of Text with Multiple Statements

Figure 2.2 performs stream insertion in multiple statements (lines 7–8), yet produces the same output as Fig. 2.1. Each stream insertion resumes displaying where the previous one stopped. Line 7 displays `Welcome` followed by a space, and because this string did not end with `\n`, line 8 begins displaying on the same line immediately following the space.

```
1  // fig02_02.cpp
2  // Displaying a line of text with multiple statements.
3  #include <iostream> // enables program to output data to the screen
4
5  // function main begins program execution
6  int main() {
7     std::cout << "Welcome ";
8     std::cout << "to C++!\n";
9  } // end function main
```

```
Welcome to C++!
```

Fig. 2.2 Displaying a line of text with multiple statements.

Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using additional newline characters, as in line 7 of Fig. 2.3. Each time the `\n` (newline) escape sequence is encountered in the output stream, the screen cursor is positioned to the beginning of the next line. To get a blank line in your output, place two newline characters back to back, as in line 7.

```
1   // fig02_03.cpp
2   // Displaying multiple lines of text with a single statement.
3   #include <iostream> // enables program to output data to the screen
4
5   // function main begins program execution
6   int main() {
7      std::cout << "Welcome\nto\n\nC++!\n";
8   } // end function main
```

```
Welcome
to

C++!
```

Fig. 2.3 Displaying multiple lines of text with a single statement.

2.4 Another C++ Program: Adding Integers

Our next program obtains two integers typed by a user at the keyboard, computes their sum and outputs the result using `std::cout`. Figure 2.4 shows the program and sample inputs and outputs. In the sample execution, the user's input is in **bold**.

```
1   // fig02_04.cpp
2   // Addition program that displays the sum of two integers.
3   #include <iostream> // enables program to perform input and output
4
5   // function main begins program execution
6   int main() {
7      // declaring and initializing variables
8      int number1{0}; // first integer to add (initialized to 0)
9      int number2{0}; // second integer to add (initialized to 0)
10     int sum{0}; // sum of number1 and number2 (initialized to 0)
11
12     std::cout << "Enter first integer: "; // prompt user for data
13     std::cin >> number1; // read first integer from user into number1
14
15     std::cout << "Enter second integer: "; // prompt user for data
16     std::cin >> number2; // read second integer from user into number2
17
18     sum = number1 + number2; // add the numbers; store result in sum
```

```
19
20      std::cout << "Sum is " << sum << "\n"; // display sum
21   } // end function main
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.4 Addition program that displays the sum of two integers.

Variable Declarations and Braced Initialization

Lines 8–10

```
int number1{0}; // first integer to add (initialized to 0)
int number2{0}; // second integer to add (initialized to 0)
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

are **declarations**—`number1`, `number2` and `sum` are the names of **variables**. These declarations specify that the variables `number1`, `number2` and `sum` are data of type `int`, meaning they will hold **integer** (whole number) values, such as 7, –11, 0 and 31914. All variables must be declared with a name and a data type.

Lines 8–10 initialize each variable to `0` by placing a value in braces (`{` and `}`) immediately following the variable's name. This is known as **braced initialization**, which was introduced in C++11. Although it's not always necessary to initialize every variable explicitly, doing so will help you avoid many kinds of problems.

Prior to C++11, lines 8–10 would have been written as:

```
int number1 = 0; // first integer to add (initialized to 0)
int number2 = 0; // second integer to add (initialized to 0)
int sum = 0; // sum of number1 and number2 (initialized to 0)
```

In legacy C++ programs, you're likely to encounter initialization statements using this older C++ coding style. In subsequent chapters, we'll discuss various benefits of braced initializers.

Declaring Multiple Variables at Once

Variables of the same type may be declared in one declaration—for example, we could have declared and initialized all three variables using a comma-separated list as follows:

```
int number1{0}, number2{0}, sum{0};
```

However, this makes the program less readable and makes it awkward to provide comments that describe each variable's purpose.

Fundamental Types

We'll soon discuss the type `double` for specifying real numbers and the type `char` for specifying character data. Real numbers are numbers with decimal points, such as 3.4, 0.0 and –11.19.
A `char` variable may hold only a single lowercase letter, uppercase letter, digit or special character (e.g., `$` or `*`). Types such as `int`, `double`, `char` and `long long` are called **fundamental types**. Fundamental-type names typically consist of one or more keywords and must appear in all lowercase letters. For a complete list of C++ fundamental types and their typical ranges, see

```
https://en.cppreference.com/w/cpp/language/types
```

Identifiers and Camel-Case Naming

A variable name (such as `number1`) may be any valid **identifier**. An identifier is a series of characters consisting of letters, digits and underscores (_) that does not begin with a digit and is not a keyword. C++ is **case sensitive**—uppercase and lowercase letters are different. So, `a1` and `A1` are different identifiers.

C++ allows identifiers of any length. Do not begin an identifier with an underscore and a capital letter or two underscores—C++ compilers use names like that for their own purposes internally.

By convention, variable-name identifiers begin with a lowercase letter, and every word in the name after the first word begins with a capital letter—e.g., `firstNumber` starts its second word, `Number`, with a capital `N`. This naming convention is known as **camel case**, because the uppercase letters stand out like a camel's humps.

Placement of Variable Declarations

Variable declarations can be placed almost anywhere in a program, but they must appear before the variables are used. For example, the declaration in line 8

```
int number1{0}; // first integer to add (initialized to 0)
```

could have been placed immediately before line 13:

```
std::cin >> number1; // read first integer from user into number1
```

the declaration in line 9:

```
int number2{0}; // second integer to add (initialized to 0)
```

could have been placed immediately before line 16:

```
std::cin >> number2; // read second integer from user into number2
```

and the declaration in line 10:

```
int sum{0}; // sum of number1 and number2 (initialized to 0)
```

could have been placed immediately before line 18:

```
sum = number1 + number2; // add the numbers; store result in sum
```

In fact, lines 10 and 18 could have been combined into the following declaration and placed just before line 20:

```
int sum{number1 + number2}; // initialize sum with number1 + number2
```

Obtaining the First Value from the User

Line 12

```
std::cout << "Enter first integer: "; // prompt user for data
```

displays `Enter first integer:` followed by a space. This message is called a **prompt** because it directs the user to take a specific action. Line 13

```
std::cin >> number1; // read first integer from user into number1
```

uses the **standard input stream object** `cin` (of namespace `std`) and the **stream extraction operator**, `>>`, to obtain a value from the keyboard.

When the preceding statement executes, the program waits for you to enter a value for variable `number1`. You respond by typing an integer (as characters), then pressing the *Enter* key (sometimes called the *Return* key) to send the characters to the program. The `cin` object converts the character representation of the number to an integer value and assigns this value to the variable `number1`. Pressing *Enter* also causes the cursor to move to the beginning of the next line on the screen.

When your program is expecting the user to enter an integer, the user could enter alphabetic characters, special symbols (like # or @) or a number with a decimal point (like 73.5), among others. In these early programs, we assume that the user enters valid data. We'll present various techniques for dealing with data-entry problems later.

Obtaining the Second Value from the User

Line 15

```
std::cout << "Enter second integer: "; // prompt user for data
```

displays `Enter second integer:` on the screen, prompting the user to take action. Line 16

```
std::cin >> number2; // read second integer from user into number2
```

obtains a value for variable `number2` from the user.

Calculating the Sum of the Values Input by the User

The assignment statement in line 18

```
sum = number1 + number2; // add the numbers; store result in sum
```

adds the values of `number1` and `number2` and assigns the result to `sum` using the **assignment operator** `=`. Most calculations are performed in assignment statements. The `=` operator and the `+` operator are **binary operators**, because each has two operands. For the `+` operator, the two operands are `number1` and `number2`. For the preceding `=` operator, the two operands are `sum` and the value of the expression `number1 + number2`. Placing spaces on either side of a binary operator makes the operator stand out and makes the program more readable.

Displaying the Result

Line 20

```
std::cout << "Sum is " << sum << "\n"; // display sum
```

displays the character string `"Sum is"` followed by the numerical value of variable `sum` and a newline.

The preceding statement outputs multiple values of different types. The stream insertion operator "knows" how to output each type of data. Using multiple stream insertion operators (`<<`) in a single statement is referred to as **concatenating**, **chaining** or **cascading stream insertion operations**.

Calculations can also be performed in output statements. We could have eliminated the variable `sum` by combining the statements in lines 18 and 20 into the statement

```
std::cout << "Sum is " << number1 + number2 << "\n";
```

The signature feature of C++ is that you can create your own data types called classes (we discuss this topic beginning in Chapter 9). You can then "teach" C++ how to input and output values of these new data types using the `>>` and `<<` operators, respectively. This is called **operator overloading**, which we explore in Chapter 11.

2.5 Arithmetic

The following table summarizes the **arithmetic operators**:

Operation	Arithmetic operator	Algebraic expression	C++ expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm or $b \cdot m$	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	`%`	$r \bmod s$	`r % s`

Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates multiplication and the **percent sign** (`%`) is the remainder operator, which we'll discuss shortly. These arithmetic operators are all binary operators.

Integer Division

Integer division in which the numerator and the denominator are integers yields an integer quotient. For example, the expression `7/4` evaluates to `1`, and the expression `17 / 5` evaluates to `3`. Any fractional part in the result of integer division is truncated—no rounding occurs.

Remainder Operator

The **remainder operator**, `%` (also called the **modulus operator**), yields the remainder after integer division and can be used only with integer operands. The expression `x % y` yields the remainder after dividing `x` by `y`. Thus, `7%4` yields `3` and `17%5` yields `2`.

Parentheses for Grouping Subexpressions

Parentheses are used in C++ expressions in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c` we write `a * (b + c)`.

Rules of Operator Precedence

C++ applies the operators in arithmetic expressions in a precise order determined by the following **rules of operator precedence**, which are generally the same as those in algebra:

1. Expressions in parentheses evaluate first. Parentheses are said to be at the "highest level of precedence." In cases of **nested** or **embedded parentheses**, such as

```
(a * (b + c))
```

expressions in the innermost pair of parentheses evaluate first.

2. Multiplication, division and remainder operations evaluate next. In an expression containing several of these operations, they're applied from left-to-right. These three operators are said to be on the same level of precedence.

3. Addition and subtraction operations evaluate last. If an expression contains several of these operations, they're applied from left-to-right. Addition and subtraction also have the same level of precedence.

Online Appendix A contains the complete operator precedence chart. **Caution:** In an expression such as `(a + b) * (c - d)`, where two sets of parentheses are not nested but appear "on the same level," the C++ Standard does not specify the order in which these parenthesized subexpressions will evaluate.

Operator Grouping

When we say that C++ applies certain operators from left-to-right, we are referring to the operators' **grouping** (sometimes called **associativity**). For example, in the expression

```
a + b + c
```

the addition operators (`+`) group from left-to-right as if we parenthesized the expression as `(a+b)+c`. Most C++ operators of the same precedence group from left-to-right. We'll see that some operators group from right-to-left.

2.6 Decision Making: Equality and Relational Operators

We now introduce C++'s **`if` statement**, which allows a program to take alternative actions based on whether a **condition** is true or false. Conditions in `if` statements can be formed by using the **relational operators** and **equality operators** in the following table:

Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
Relational operators			
>	>	`x > y`	`x` is greater than `y`
<	<	`x < y`	`x` is less than `y`
≥	>=	`x >= y`	`x` is greater than or equal to `y`

Algebraic relational or equality operator	C++ relational or equality operator	Sample C++ condition	Meaning of C++ condition
≤	<=	`x <= y`	`x` is less than or equal to `y`
Equality operators			
=	==	`x == y`	`x` is equal to `y`
≠	!=	`x != y`	`x` is not equal to `y`

The relational operators all have the same level of precedence and group from left-to-right. The equality operators both have the same level of precedence, which is lower than that of the relational operators, and group from left-to-right.

Err Reversing the order of the pair of symbols in the operators `!=`, `>=` and `<=` (by writing them as `=!`, `=>` and `=<`, respectively) is normally a syntax error. In some cases, writing `!=` as `=!` will not be a syntax error, but almost certainly it will be a logic error that has an effect at execution time. You'll understand why when we cover logical operators in Section 4.11.

Confusing `==` and `=`

Err Confusing the equality operator `==` with the assignment operator `=` results in logic errors. We like to read the equality operator as "is equal to" or "double equals" and the assignment operator as "gets" or "gets the value of" or "is assigned the value of." Confusing these operators may not necessarily cause an easy-to-recognize syntax error, but it may cause subtle logic errors. Compilers generally warn about this.

Using the `if` Statement

Figure 2.5 uses six `if` statements to compare two integers input by the user. If a given `if` statement's condition is true, the output statement in the body of that `if` statement executes. If the condition is false, the output statement in the body does not execute.

```
1  // fig02_05.cpp
2  // Comparing integers using if statements, relational operators
3  // and equality operators.
4  #include <iostream> // enables program to perform input and output
5
6  using std::cout; // program uses cout
7  using std::cin; // program uses cin
8
9  // function main begins program execution
10 int main() {
11    int number1{0}; // first integer to compare (initialized to 0)
12    int number2{0}; // second integer to compare (initialized to 0)
13
```

```
14      cout << "Enter two integers to compare: "; // prompt user for data
15      cin >> number1 >> number2; // read two integers from user
16
17      if (number1 == number2) {
18         cout << number1 << " == " << number2 << "\n";
19      }
20
21      if (number1 != number2) {
22         cout << number1 << " != " << number2 << "\n";
23      }
24
25      if (number1 < number2) {
26         cout << number1 << " < " << number2 << "\n";
27      }
28
29      if (number1 > number2) {
30         cout << number1 << " > " << number2 << "\n";
31      }
32
33      if (number1 <= number2) {
34         cout << number1 << " <= " << number2 << "\n";
35      }
36
37      if (number1 >= number2) {
38         cout << number1 << " >= " << number2 << "\n";
39      }
40   } // end function main
```

```
Enter two integers to compare: 3 7
3 != 7
3 < 7
3 <= 7
Enter two integers to compare: 22 12
22 != 12
22 > 12
22 >= 12
Enter two integers to compare: 7 7
7 == 7
7 <= 7
7 >= 7
```

Fig. 2.5 Comparing integers using `if` statements, relational operators and equality operators.

`using` Declarations

Lines 6–7

```
using std::cout; // program uses cout
using std::cin; // program uses cin
```

are **`using` declarations** that eliminate the need to repeat the `std::` prefix as we did in earlier programs. We can now write `cout` instead of `std::cout` and `cin` instead of `std::cin` in the remainder of the program.

`using` Directive

In place of lines 6–7, many programmers prefer the **`using` directive**

```
using namespace std;
```

which enables your program to use names from the `std` namespace without the `std::` qualification. In the early chapters, we'll use this directive in our programs to simplify the code.[2]

2. In online Chapter 19, we'll discuss some disadvantages of `using` directives in large-scale systems.

Variable Declarations and Reading the Inputs from the User

Lines 11–12

```
int number1{0}; // first integer to compare (initialized to 0)
int number2{0}; // second integer to compare (initialized to 0)
```

declare the variables used in the program and initialize them to `0`.

Line 15

```
cin >> number1 >> number2; // read two integers from user
```

uses cascaded stream extraction operations to input two integers. Recall that we're allowed to write `cin` (instead of `std::cin`) because of line 7. This statement first reads a value into `number1`, then into `number2`.

Comparing Numbers

The `if` statement in lines 17–19

```
if (number1 == number2) {
   cout << number1 << " == " << number2 << "\n";
}
```

determines whether the values of variables `number1` and `number2` are equal. If so, the `cout` statement displays a line of text indicating that the numbers are equal. For each condition that is `true` in the remaining `if` statements starting in lines 21, 25, 29, 33 and 37, the corresponding `cout` statement displays an appropriate line of text.

Braces and Blocks

Each `if` statement in Fig. 2.5 contains a single body statement that's indented to enhance readability. Also, notice that we've enclosed each body statement in a pair of braces, `{ }`, creating what's called a **compound statement** or a **block**.

Err You don't need to use braces around single-statement bodies, but you must include the braces around multiple-statement bodies. Forgetting to enclose multiple-statement bodies in braces

leads to errors. To avoid errors, as a rule, always enclose an `if` statement's body statement(s) in braces.

Common Logic Error: Placing a Semicolon after a Condition

Err Placing a semicolon immediately after the right parenthesis of the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement now becomes a statement in sequence with the `if` statement and always executes, often causing the program to produce incorrect results. Most compilers will issue a warning for this logic error.

Splitting Lengthy Statements

A lengthy statement may be spread over several lines. If you must do this, choose meaningful breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, it's a good practice to indent all subsequent lines.

Operator Precedence and Grouping

With the exception of the assignment operator `=`, all the operators presented in this chapter group from left-to-right. Assignments (`=`) group from right-to-left. So, an expression such as `x = y = 0` evaluates as if it had been written `x = (y = 0)`, which first assigns `0` to `y`, then assigns the result of that assignment (that is, `0`) to `x`.

Refer to the complete operator-precedence chart in online Appendix A when writing expressions containing many operators. Confirm that the operators in the expression are performed in the order you expect. If you're uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you'd do in an algebraic expression.

2.7 Objects Natural: Creating and Using Objects of Standard-Library Class `string`

Throughout this book, we emphasize using preexisting valuable classes from the C++ standard library and various open-source libraries from the C++ open-source community. You'll focus on knowing what libraries are out there, choosing the ones you'll need for your applications, creating objects from existing library classes and making those objects exercise their capabilities. By Objects Natural, we mean that you'll be able to program with powerful objects before you learn to create custom classes.

You've already worked with C++ objects—specifically the `cout` and `cin` objects, which encapsulate the mechanisms for output and input, respectively. These objects were created for you behind the scenes using classes from the header `<iostream>`. In this section, you'll create and interact with objects of the C++ standard library's **`string`**[3] class.

3. You'll learn additional `string` capabilities in subsequent chapters. Chapter 8 discusses class `string` in detail, test-driving many more of its member functions.

Test-Driving Class `string`

Classes cannot execute by themselves. A `Person` object can drive a `Car` object by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without knowing how the car's internal mechanisms work. Similarly, the `main` function can "drive" a `string` object by calling its member functions—without knowing how the class is implemented. In this sense, `main` in the following program is referred to as a **driver program**. Figure 2.6's `main` function test-drives several `string` member functions.

```
1   // fig02_06.cpp
2   // Standard library string class test program.
3   #include <iostream>
4   #include <string>
5   using namespace std;
6
7   int main() {
8      string s1{"happy"};
9      string s2{" birthday"};
10     string s3; // creates an empty string
11
12     // display the strings and show their lengths
13     cout << "s1: \"" << s1 << "\"; length: " << s1.length()
14        << "\ns2: \"" << s2 << "\"; length: " << s2.length()
15        << "\ns3: \"" << s3 << "\"; length: " << s3.length();
16
17     // compare strings with == and !=
18     cout << "\n\nThe results of comparing s2 and s1:" << boolalpha
19        << "\ns2 == s1: " << (s2 == s1)
20        << "\ns2 != s1: " << (s2 != s1);
21
22     // test string member function empty
23     cout << "\n\nTesting s3.empty():\n";
24
25     if (s3.empty()) {
26        cout << "s3 is empty; assigning to s3;\n";
27        s3 = s1 + s2; // assign s3 the result of concatenating s1 and s2
28        cout << "s3: \"" << s3 << "\"";
29     }
30
31     // testing new C++20 string member functions
32     cout << "\n\ns1 starts with \"ha\": " << s1.starts_with("ha") << "\n";
33     cout << "s2 starts with \"ha\": " << s2.starts_with("ha") << "\n";
34     cout << "s1 ends with \"ay\": " << s1.ends_with("ay") << "\n";
35     cout << "s2 ends with \"ay\": " << s2.ends_with("ay") << "\n";
36  }
```

```
s1: "happy"; length: 5
s2: " birthday"; length: 9
s3: ""; length: 0

The results of comparing s2 and s1:
s2 == s1: false
s2 != s1: true

Testing s3.empty():
s3 is empty; assigning to s3;
```

```
s3: "happy birthday"

s1 starts with "ha": true
s2 starts with "ha": false
s1 ends with "ay": false
s2 ends with "ay": true
```

Fig. 2.6 Standard library `string` class test program.

Instantiating Objects

Typically, you cannot call a member function of a class until you create an object of that class[4]—also called instantiating an object. Lines 8–10 create three `string` objects:

4. You'll see in Section 9.20 that you can call a class's `static` member functions without creating an object of that class.

- `s1` is initialized with a copy of the string literal `"happy"`,
- `s2` is initialized with a copy of the string literal `" birthday"`, and
- `s3` is initialized by default to the **empty `string`** (that is, `""`).

When we declare `int` variables, as we did earlier, the compiler knows what `int` is—it's a fundamental type that's built into C++. In lines 8–10, however, the compiler does not know in advance what type `string` is—it's a class type from the C++ standard library.

When packaged properly, classes can be reused by other programmers. This is one of the most significant benefits of working with object-oriented programming languages like C++ that have rich libraries of powerful prebuilt classes. For example, you can reuse the C++ standard library's classes in any program by including the appropriate headers—in this case, the **`<string>` header** (line 4). The name `string`, like the name `cout`, belongs to namespace `std`.

`string` Member Function `length`

Lines 13–15 output each `string` and its length. The `string` class's **`length` member function** returns the number of characters stored in a particular `string` object. In line 13, the expression

```
s1.length()
```

returns `s1`'s length by calling the object's `length` member function. To call this member function for a specific object, you specify the object's name (`s1`), followed by the **dot operator** (`.`), then the member function name (`length`) and a set of parentheses. *Empty* parentheses indicate that `length` does not require any additional information to perform its task. Soon, you'll see that some member functions require additional information called arguments to perform their tasks.

From `main`'s view, when the `length` member function is called:

1. The program transfers execution from the call (line 13 in `main`) to member function `length`. Because `length` was called via the `s1` object, `length` "knows" which object's data to manipulate.
2. Next, member function `length` performs its task—that is, it returns `s1`'s length to line 13 where the function was called. The `main` function does not know the details of how `length` performs its task, just as the driver of a car doesn't know the details of how engines, transmissions, steering mechanisms and brakes are implemented.
3. The `cout` object displays the number of characters returned by member function `length`, then the program continues executing, displaying the `string`s `s2` and `s3` and their lengths.

Comparing `string` Objects with the Equality Operators

Like numbers, `string`s can be compared with one another. Lines 18–20 compare `s2` to `s1` using the equality operators—`string` comparisons are case sensitive.[5]

---

5. In Chapter 8, you'll see that strings perform lexicographical comparisons using the numerical values of the characters in each string.

Normally, when you output a condition's value, C++ displays `0` for false or `1` for true. The stream manipulator **`boolalpha`** (line 18) from the `<iostream>` header tells the output stream to display condition values as the words `false` or `true`.

`string` Member Function `empty`

Line 25 calls `string` member function **`empty`**, which returns `true` if the `string` is empty—that is, the length of the `string` is `0`. Otherwise, `empty` returns `false`. The object `s3` was initialized by default to the empty string, so it is indeed empty, and the body of the `if` statement will execute.

`string` Concatenation and Assignment

Line 27 assigns a new value to `s3` produced by "adding" the strings `s1` and `s2` using the `+` operator—this is known as **`string` concatenation**. After the assignment, `s3` contains the characters of `s1` followed by the characters of `s2`—`"happy birthday"`. Line 28 outputs `s3` to demonstrate that the assignment worked correctly.

C++20 `string` Member Functions `starts_with` and `ends_with`

20 Lines 32–35 demonstrate new C++20 `string` member functions **`starts_with`** and **`ends_with`**, which return `true` if the `string` starts with or ends with a specified substring, respectively; otherwise, they return `false`. Lines 32 and 33 show that `s1` starts with `"ha"`, but `s2` does not. Lines 34 and 35 show that `s1` does not end with `"ay"` but `s2` does.

2.8 Wrap-Up

We presented many important basic features of C++ in this chapter, including displaying data on the screen, inputting data from the keyboard and declaring variables of fundamental types. In particular, you learned to use the output stream object `cout` and the input stream object `cin` to build simple interactive programs. We declared and initialized variables and used arithmetic operators to perform calculations. We discussed the order in which C++ applies operators (i.e., the rules of operator precedence), as well as the grouping of the operators (also called the associativity of the operators). You saw how C++'s `if` statement allows a program to make decisions. We introduced the equality and relational operators, which we used to form conditions in `if` statements.

Finally, we introduced our "Objects Natural" approach to learning C++ by creating objects of the C++ standard-library class `string` and interacting with them using equality operators and `string` member functions. In subsequent chapters, you'll create and use many objects of existing classes to accomplish significant tasks with minimal amounts of code. Then, in Chapters 9–11, you'll create your own custom classes. You'll see that C++ enables you to "craft valuable classes." In the next chapter, we begin our introduction to control statements, which specify the order in which a program's actions are performed.

3. Control Statements: Part 1

Objectives

In this chapter, you'll:

- Use the `if` and `if`...`else` selection statements to choose between alternative actions.
- Use the `while` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Use nested control statements.
- Use the compound assignment operators and the increment and decrement operators.
- Learn why fundamental data types are not portable.
- Continue our Objects Natural approach with a case study on creating and manipulating integers as large as you want them to be.
- Use C++20's new text-formatting capabilities, which are more concise and more powerful than those in earlier C++ versions.

Outline

3.1 Introduction
3.2 Control Structures
3.2.1 Sequence Structure
3.2.2 Selection Statements
3.2.3 Iteration Statements
3.2.4 Summary of Control Statements
3.3 `if` Single-Selection Statement
3.4 `if`...`else` Double-Selection Statement
3.4.1 Nested `if`...`else` Statements
3.4.2 Blocks
3.4.3 Conditional Operator (`?:`)
3.5 `while` Iteration Statement
3.6 Counter-Controlled Iteration
3.6.1 Implementing Counter-Controlled Iteration
3.6.2 Integer Division and Truncation
3.7 Sentinel-Controlled Iteration
3.7.1 Implementing Sentinel-Controlled Iteration
3.7.2 Converting Between Fundamental Types Explicitly and Implicitly
3.7.3 Formatting Floating-Point Numbers
3.8 Nested Control Statements
3.8.1 Problem Statement
3.8.2 Implementing the Program
3.8.3 Preventing Narrowing Conversions with Braced Initialization
3.9 Compound Assignment Operators
3.10 Increment and Decrement Operators
3.11 Fundamental Types Are Not Portable
3.12 Objects Natural Case Study: Arbitrary-Sized Integers
3.13 C++20: Text Formatting with Function `format`
3.14 Wrap-Up

3.1 Introduction

In this chapter and the next, we present the theory and principles of structured programming. The concepts presented here are crucial in building classes and manipulating objects. We discuss the `if` statement in additional detail and introduce the `if`...`else` and `while` statements. We also introduce the compound assignment operators and the increment and decrement operators.

We discuss why the fundamental types are not portable. We continue our Objects Natural approach with a case study on arbitrary-sized integers that can represent values beyond the ranges of integers supported by computer hardware.

20 We begin introducing C++20's new text-formatting capabilities, which are based on those in Python, Microsoft's .NET languages (like C# and Visual Basic) and Rust.[1] The C++20 capabilities are more concise and more powerful than those in earlier C++ versions.

1. Victor Zverovich, "Text Formatting," July 16, 2019. Accessed November 11, 2021. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html.

3.2 Control Structures

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of many problems experienced by software development groups. The blame was pointed at the **`goto` statement** (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program.

The research of Böhm and Jacopini[2] had demonstrated that programs could be written without any `goto` statements. The challenge for programmers of the era was to shift their styles to "`goto`-less programming." The term **structured programming** became almost synonymous with "`goto` elimination." The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug-free in the first place.

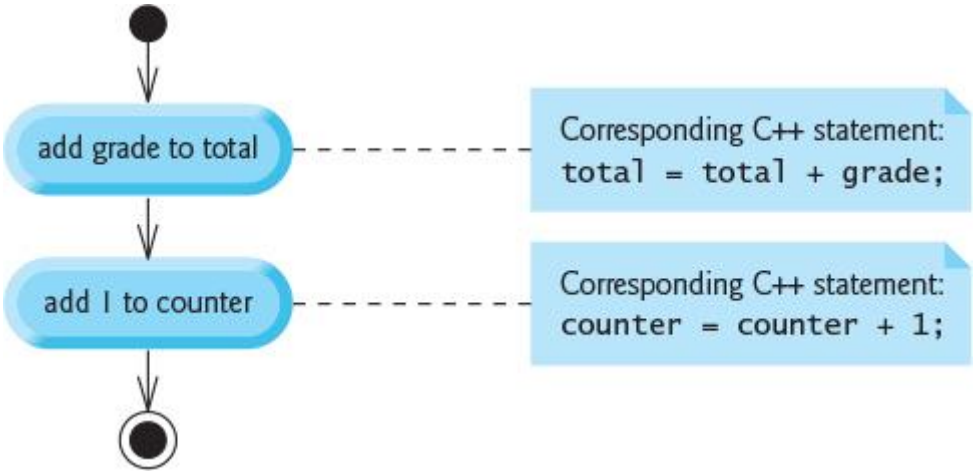
2. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

Böhm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures—the **sequence structure**, the **selection structure** and the **iteration structure**. We'll discuss how C++ implements each of these.

3.2.1 Sequence Structure

The sequence structure is built into C++. Unless directed otherwise, statements execute one after the other in the order they appear in the program—that is, in sequence. The following UML[3] **activity diagram** illustrates a typical sequence structure in which two calculations are performed in order:

3. We use the UML in this chapter and Chapter 4 to show the flow of control in control statements, then use UML again in Chapters 9–10 when we present custom class development.


add grade to total
Corresponding C++ statement:
total = total + grade;
add 1 to counter
Corresponding C++ statement:
counter = counter + 1;

C++ lets you have as many actions as you want in a sequence structure. As you'll soon see, anywhere you may place a single action, you may place several actions in sequence.

An activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, like the sequence structure in the preceding diagram. Activity diagrams are composed of symbols, such as **action-state symbols** (rectangles with their left and right sides replaced with outward arcs), **diamonds** and **small circles**. These symbols are connected by **transition arrows**, representing the activity's flow—that is, the order in which the actions should occur.

The preceding sequence-structure activity diagram contains two **action states**, each containing an **action expression**—for example, "add grade to total" or "add 1 to counter"—that specifies a particular action to perform. The arrows in the activity diagram represent **transitions**, which indicate the order in which the actions represented by the action states occur.

The **solid circle** at the top of the activity diagram represents the **initial state**—the beginning of the workflow before the program performs the modeled actions. The **solid circle surrounded by a hollow circle** at the bottom of the diagram represents the **final state**—that is, the end of the workflow after the program performs its actions.

The sequence-structure activity diagram also includes rectangles with the upper-right corners folded over. These are UML **notes** (like comments in C++)—explanatory remarks that describe the purpose of symbols in the diagram. A **dotted line** connects each note with the element it describes. This diagram's UML notes show how the diagram relates to the C++ code for each action state. Activity diagrams usually do not show the C++ code.

3.2.2 Selection Statements

C++ has three types of **selection statements**. The `if` statement performs (selects) an action (or group of actions) if a condition is true, or skips it if the condition is false. The `if`...`else` statement performs an action (or group of actions) if a condition is true and performs a different action (or group of actions)

if the condition is false. The `switch` statement (Chapter 4) performs one of many different actions (or groups of actions), depending on the value of an expression.

The `if` statement is called a **single-selection statement** because it selects or ignores a single action (or group of actions). The `if`...`else` statement is called a **double-selection statement** because it selects between two different actions (or groups of actions). The `switch` statement is called a **multiple-selection statement** because it selects among many different actions (or groups of actions).

3.2.3 Iteration Statements

C++ provides four **iteration statements**—also called **repetition statements** or **looping statements**—for performing statements repeatedly while a **loop-continuation condition** remains true. The iteration statements are the `while`, `do`...`while`, `for` and range-based `for`. The `while` and `for` statements perform their action (or group of actions) zero or more times. If the loop-continuation condition is initially false, the action (or group of actions) does not execute. The `do`...`while` statement performs its action (or group of actions) one or more times. Chapter 4 presents the `do`...`while` and `for` statements. Chapter 6 presents the range-based `for` statement.

Keywords

Each of the words `if`, `else`, `switch`, `while`, `do` and `for` is a C++ keyword. Keywords cannot be used as identifiers, such as variable names, and contain only lowercase letters (and sometimes underscores). The following table shows the complete list of C++ keywords:

C++ keywords				
alignas	alignof	and	and_eq	asm
auto	bitand	bitor	bool	break
case	catch	char	char16_t	char32_t
class	compl	const	const_cast	constexpr
continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	final	float
for	friend	goto	if	import
inline	int	long	module	mutable
namespace	new	noexcept	not	not_eq
nullptr	operator	or	or_eq	override
private	protected	public	register	reinterpret_cast

C++ keywords

return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template
this	thread_local	throw	true	try
typedef	typeid	typename	union	unsigned
using	void	volatile	virtual	wchar_t
while	xor	xor_eq		

20 Keywords new in C++20

char8_t	concept	consteval	constinit	co_await
co_return	co_yield	requires		

3.2.4 Summary of Control Statements

C++ has only three kinds of control structures, which from this point forward, we refer to as control statements:

- sequence,
- selection (`if`, `if`...`else` and `switch`) and
- iteration (`while`, `do`...`while`, `for` and range-based `for`).

You form every program by combining these statements as appropriate for the algorithm you're implementing. We can model each control statement as an activity diagram. Each diagram contains an initial state and a final state representing a control statement's entry point and exit point, respectively. **Single-entry/single-exit control statements** make it easy to build readable programs—we simply connect the exit point of one to the entry point of the next using **control-statement stacking**. There's only one other way in which you may connect control statements—**control-statement nesting**, in which one control statement appears inside another. Thus, algorithms in C++ programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity.

3.3 `if` Single-Selection Statement

We introduced the `if` single-selection statement briefly in Section 2.6. Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The following C++ statement determines whether the condition `studentGrade >= 60` is true:

```
if (studentGrade >= 60) {
   cout << "Passed";
}
```

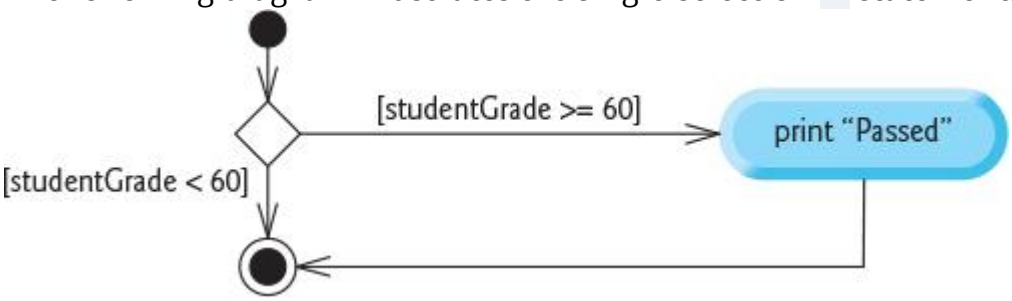
If so, `"Passed"` is printed, and the next statement in order is performed. If the condition is false, the output statement is ignored, and the next statement in order is performed. The indentation of the second line of this selection statement is optional but recommended for program clarity.

`bool` Data Type

In Chapter 2, you created conditions using the relational or equality operators. Actually, any expression that evaluates to zero or nonzero can be used as a condition. Zero is treated as false, and nonzero is treated as true. C++ also provides the data type **`bool`** for Boolean variables that can hold only the values **`true`** and **`false`**—each is a C++ keyword. The compiler can implicitly convert `true` to `1` and `false` to `0`.

UML Activity Diagram for an `if` Statement

The following diagram illustrates the single-selection `if` statement.



This figure contains the most important symbol in an activity diagram—the diamond, or **decision symbol**, which indicates that a decision is to be made. The workflow continues along a path determined by the symbol's associated **guard conditions**, which can be true or false. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets next to the arrow). If a guard condition is true, the workflow enters the action state to which the transition arrow points. The diagram shows that if the grade is greater than or equal to 60 (i.e., the condition is true), the program prints "Passed" then transitions to the activity's final state. If the grade is less than 60 (i.e., the condition is false), the program immediately transitions to the final state without displaying a message. The `if` statement is a single-entry/single-exit control statement.

3.4 `if` ...`else` Double-Selection Statement

The `if` single-selection statement performs an indicated action only when the condition is true. The **`if`...`else` double-selection statement** allows you to specify an action to perform when the condition is true and another action when the condition is false. For example, the following C++ statement prints `"Passed"` if `studentGrade >= 60`, but prints `"Failed"` if it's less than 60:

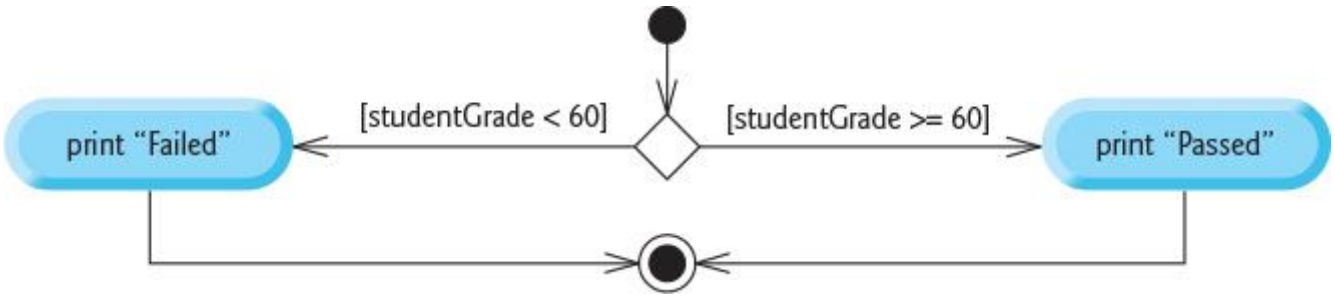
```
if (studentGrade >= 60) {
   cout << "Passed";
}
else {
   cout << "Failed";
}
```

In either case, after printing occurs, the next statement in sequence is performed.

The body of the `else` also is indented. Whatever indentation convention you choose should be applied consistently throughout your programs.

UML Activity Diagram for an `if`...`else` Statement

The following diagram illustrates the flow of control in the preceding `if`...`else` statement:



3.4.1 Nested `if...else` Statements

A program can test multiple cases by placing `if...else` statements inside other `if...else` statements to create **nested `if...else` statements**. For example, the following nested `if...else` prints `"A"` for exam grades greater than or equal to 90, `"B"` for grades 80 to 89, `"C"` for grades 70 to 79, `"D"` for grades 60 to 69 and `"F"` for all other grades. We use shading to highlight the nesting.

```
if (studentGrade >= 90) {
   cout << "A";
}
else {
   if (studentGrade >= 80) {
      cout << "B";
   }
   else {
      if (studentGrade >= 70) {
         cout << "C";
      }
      else {
         if (studentGrade >= 60) {
            cout << "D";
         }
         else {
            cout << "F";
         }
      }
   }
}
```

If variable `studentGrade` is greater than or equal to 90, the first four conditions in the nested `if...else` statement will be true, but only the statement in the `if` part of the first `if...else` statement will execute. After that statement executes, the `else` part of the "outermost" `if...else` statement is skipped. The preceding nested `if...else` statement also can be written in the following form, which is identical but uses fewer braces, less spacing and indentation:

```
if (studentGrade >= 90) {
   cout << "A";
}
else if (studentGrade >= 80) {
   cout << "B";
}
else if (studentGrade >= 70) {
```

```
   cout << "C";
}
else if (studentGrade >= 60) {
   cout << "D";
}
else {
   cout << "F";
}
```

This form avoids deep indentation of the code to the right, which can force lines to wrap. Throughout the text, we always enclose control-statement bodies in braces (`{` and `}`), which avoids a logic error called the "dangling-`else`" problem.

3.4.2 Blocks

The `if` statement expects only one statement in its body. To include several statements in an `if`'s or `else`'s body, enclose the statements in braces. It's good practice always to use the braces. Statements in a pair of braces (such as a control statement's or function's body) form a **block**. A block can be placed anywhere in a function that a single statement can be placed.

The following example includes a block of multiple statements in an `if`...`else` statement's `else` part:

```
if (studentGrade >= 60) {
   cout << "Passed";
}
else {
   cout << "Failed\n";
   cout << "You must retake this course.";
}
```

If `studentGrade` is less than 60, the program executes both statements in the body of the `else` and prints

```
Failed
You must retake this course.
```

Without the braces surrounding the two statements in the `else` clause, the statement

```
cout << "You must retake this course.";
```

would be outside the body of the `else` part of the `if`...`else` statement and would execute regardless of whether the `studentGrade` was less than 60—a logic error.

Empty Statement

Just as a block can be placed anywhere a single statement can be placed, it's also possible to have an **empty statement**, which is simply a semicolon (`;`) where a statement typically would be. An empty statement has no effect.

3.4.3 Conditional Operator (?:)

C++ provides the **conditional operator** (`?:`), which can be used in place of an `if`...`else` statement. This can make your code shorter and clearer. The conditional operator is C++'s only **ternary operator** (i.e.,

an operator that takes three operands). Together, the operands and the `?:` symbol form a **conditional expression.** For example, the following statement prints the conditional expression's value:

```
cout << (studentGrade >= 60 ? "Passed":"Failed");
```

The operand to the left of the `?` is a condition. The second operand (between the `?` and `:`) is the conditional expression's value if the condition is true. The operand to the right of the `:` is the conditional expression's value if the condition is false. The conditional expression in this statement evaluates to the string `"Passed"` if the condition

```
studentGrade >= 60
```

is true and to the string `"Failed"` if it's false. Thus, this statement with the conditional operator performs essentially the same function as the first `if`...`else` statement in Section 3.4. The precedence of the conditional operator is low, so the entire conditional expression is normally placed in parentheses.

3.5 `while` Iteration Statement

An iteration statement allows you to specify that a program should repeat an action while some condition remains true.

As an example of C++'s **`while` iteration statement**, consider a program segment that finds the first power of 3 larger than 100. After the following `while` statement executes, the variable `product` contains the result:

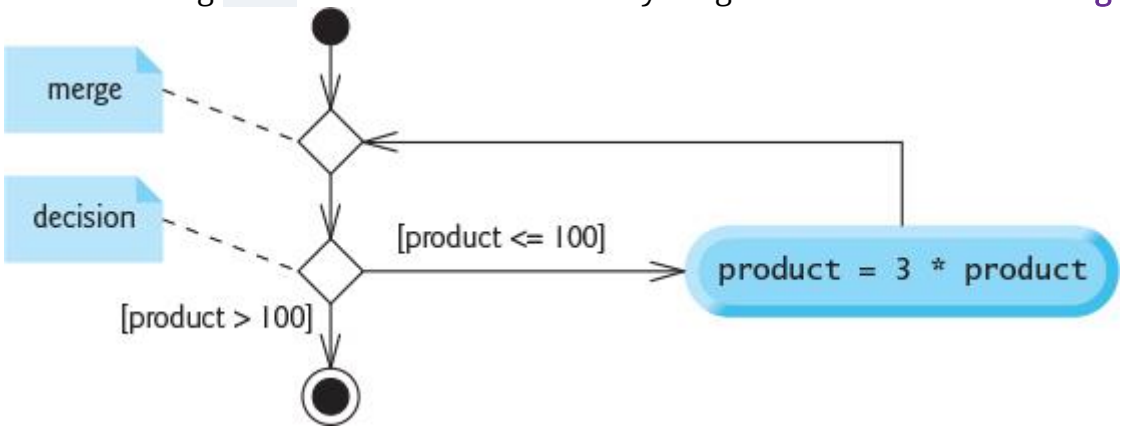
```
int product{3};

while (product <= 100) {
   product = 3 * product;
}
```

Each iteration of the `while` statement multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively. When `product` becomes 243, `product <= 100` becomes false. This terminates the iteration, so the final value of `product` is 243. At this point, program execution continues with the next statement after the `while` statement.

UML Activity Diagram for a `while` Statement

The following `while` statement UML activity diagram introduces the **merge symbol**:



The UML represents both the merge symbol and the decision symbol as diamonds. The merge symbol joins two flows of activity into one. In this diagram, the merge symbol joins the transitions from the

initial state and the action state, so they both flow into the decision that determines whether the loop should begin (or continue) executing.

You can distinguish the decision and merge symbols by the number of incoming and outgoing transition arrows. A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it to indicate possible transitions from that decision. Also, each arrow pointing out of a decision symbol has a guard condition. A merge symbol has two or more transition arrows pointing to it and only one pointing from it to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.

3.6 Counter-Controlled Iteration

Consider the following problem statement:

A class of ten students took a quiz. The grades (integers in the range 0–100) for this quiz are available to you. Determine the class average on the quiz.

The class average is equal to the sum of the grades divided by the number of students. The program must input each grade, total all the grades entered, perform the averaging calculation and print the result.

We use **counter-controlled iteration** to input the grades one at a time. This technique uses a counter to control the number of times a set of statements will execute. In this example, iteration terminates when the counter exceeds 10.

3.6.1 Implementing Counter-Controlled Iteration

In Fig. 3.1, the `main` function calculates the class average with counter-controlled iteration. It allows the user to enter 10 grades, then calculates and displays the average.

```
1  fig03_01.cpp
2  // Solving the class-average problem using counter-controlled iteration.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // initialization phase
8     int total{0}; // initialize sum of grades entered by the user
9     int gradeCounter{1}; // initialize grade # to be entered next
10
11    // processing phase uses counter-controlled iteration
12    while (gradeCounter <= 10) { // loop 10 times
13       cout << "Enter grade: "; // prompt
14       int grade;
15       cin >> grade; // input next grade
16       total = total + grade; // add grade to total
17       gradeCounter = gradeCounter + 1; // increment counter by 1
```

```
18      }
19
20      // termination phase
21      int average{total / 10}; // int division yields int result
22
23      // display total and average of grades
24      cout << "\nTotal of all 10 grades is " << total;
25      cout << "\nClass average is " << average << "\n";
26   }
```

```
Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

Total of all 10 grades is 846
Class average is 84
```

Fig. 3.1 Solving the class-average problem using counter-controlled iteration.

Local Variables in `main`

Lines 8, 9, 14 and 21 declare `int` variables `total`, `gradeCounter`, `grade` and `average`, respectively. Variable `grade` stores the user input. A variable declared in a block (such as a function's body) is a local variable that can be used only from the line of its declaration to the closing right brace of the block. A local variable's declaration must appear before the variable is used. Variable `grade`—declared in the body of the `while` loop—can be used only in that block.

Initializing Variables `total` and `gradeCounter`

Lines 8–9 declare and initialize `total` to `0` and `gradeCounter` to `1`. These initializations occur before the variables are used in calculations.

Reading 10 Grades from the User

The `while` statement (lines 12–18) continues iterating as long as `gradeCounter`'s value is less than or equal to 10. Line 13 displays the prompt `"Enter grade: "`. Line 15 inputs the grade entered by the user and assigns it to variable `grade`. Then line 16 adds the new `grade` entered by the user to the `total` and assigns the result to `total`, replacing its previous value. Line 17 adds `1` to `gradeCounter` to indicate that the program has processed a grade and is ready to input the next grade from the user. Incrementing `gradeCounter` eventually causes it to exceed 10, which terminates the loop.

Calculating and Displaying the Class Average

When the loop terminates, line 21 performs the averaging calculation in the `average` variable's initializer. Line 24 displays the text `"Total of all 10 grades is "` followed by variable `total`'s value. Then, line 25 displays the text `"Class average is "` followed by `average`'s value. When execution reaches line 26, the program terminates.

3.6.2 Integer Division and Truncation

This example's average calculation produces an `int` result. The program's sample execution shows that the sum of the grades is 846—when divided by 10, this should yield 84.6. Numbers like 84.6 containing decimal points are **floating-point numbers**. However, in the class-average program, `total / 10` produces the integer 84 because `total` and `10` are both integers. Dividing two integers results in **integer division**—any fractional part of the calculation is truncated. The next section shows how to obtain a floating-point result from the averaging calculation. For example, 7 / 4 yields 1.75 in conventional arithmetic but truncates to 1 in integer arithmetic rather than rounding to 2.

3.7 Sentinel-Controlled Iteration

Let's generalize Section 3.6's class-average problem. Consider the following problem:
Develop a class-averaging program that processes grades for an arbitrary number of students each time it's run.

In the previous class-average example, the problem statement specified the number of students, so the number of grades (10) was known in advance. Here, we do not know how many grades the user will enter during the program's execution. The program must process an *arbitrary* number of grades.

One way to solve this problem is to use a **sentinel value** (also called a **signal value**, a **dummy value** or a **flag value**) to indicate "end of data entry." The user enters grades until all legitimate grades have been entered. The user then enters the sentinel value to indicate that no more grades will be entered.

You must choose a sentinel value that cannot be confused with an acceptable input value. Grades on a quiz are non-negative integers, so –1 is an acceptable sentinel value for this problem. Thus, a run of the class-averaging program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1. The program would then compute and print the class average for the grades 95, 96, 75, 74 and 89; since –1 is the sentinel value, it should not enter into the averaging calculation.

It's possible the user could enter –1 before entering grades, in which case the number of grades will be zero. We must test for this case before calculating the class average. According to the C++ standard, the result of division by zero in floating-point arithmetic is undefined. When performing division (`/`) or remainder (`%`) calculations in which the right operand could be zero, test for this and handle it (e.g., display an error message) rather than allowing the calculation to proceed.

3.7.1 Implementing Sentinel-Controlled Iteration

Figure 3.2 implements sentinel-controlled iteration. Although each grade entered by the user is an integer, the average calculation will likely produce a floating-point number, which an `int` cannot represent. C++ provides data types `float`, `double` and `long double` to store floating-point numbers in memory. The primary difference between these types is that `double` variables typically store numbers with larger magnitude and finer detail than `float`—that is, more digits to the right of the decimal point, which is also known as the number's **precision**. Similarly, `long double` stores values with larger magnitude and more precision than `double`. We say more about floating-point types in Chapter 4.

```
1   // fig03_02.cpp
```

```
2   // Solving the class-average problem using sentinel-controlled iteration.
3   #include <iostream>
4   #include <iomanip> // parameterized stream manipulators
5   using namespace std;
6
7   int main() {
8      // initialization phase
9      int total{0}; // initialize sum of grades
10     int gradeCounter{0}; // initialize # of grades entered so far
11
12     // processing phase
13     // prompt for input and read grade from user
14     cout << "Enter grade or -1 to quit: ";
15     int grade;
16     cin >> grade;
17
18     // loop until sentinel value is read from user
19     while (grade != -1) {
20        total = total + grade; // add grade to total
21        gradeCounter = gradeCounter + 1; // increment counter
22
23        // prompt for input and read next grade from user
24        cout << "Enter grade or -1 to quit: ";
25        cin >> grade;
26     }
27
28     // termination phase
29     // if user entered at least one grade...
30     if (gradeCounter != 0) {
31        // use number with decimal point to calculate average of grades
32        double average{static_cast<double>(total) / gradeCounter};
33
34        // display total and average (with two digits of precision)
35        cout << "\nTotal of the " << gradeCounter
36           << " grades entered is " << total;
37        cout << setprecision(2) << fixed;
38        cout << "\nClass average is " << average << "\n";
39     }
40     else { // no grades were entered, so output appropriate message
41        cout << "No grades were entered\n";
42     }
43  }
```

```
Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of the 3 grades entered is 257
Class average is 85.67
```

Fig. 3.2 Solving the class-average problem using sentinel-controlled iteration.

Recall that integer division produces an integer result. This program introduces a **cast operator** to force the average calculation to produce a floating-point result. This program also stacks control

statements in sequence—the `while` statement is followed in sequence by an `if`...`else` statement. Much of this program's code is identical to Fig. 3.1, so we concentrate on only the new concepts.

Program Logic for Sentinel-Controlled Iteration vs. Counter-Controlled Iteration

Line 10 initializes `gradeCounter` to `0` because no grades have been entered yet. Remember that this program uses sentinel-controlled iteration to input the grades. The program increments `gradeCounter` only when the user enters a valid grade. Line 32 declares `double` variable `average`, which stores the calculated class average as a floating-point number.

Compare the program logic for sentinel-controlled iteration in this program with that for counter-controlled iteration in Fig. 3.1. In counter-controlled iteration, each iteration of the `while` statement (lines 12–18 of Fig. 3.1) reads a value from the user for the specified number of iterations. In sentinel-controlled iteration, the program prompts for and reads the first value (lines 14 and 16 of Fig. 3.2) before reaching the `while`. This value determines whether the flow of control should enter the `while`'s body. If the condition is false, the user entered the sentinel value, so no grades were entered, and the body does not execute. If the condition is true, the body begins execution, and the loop adds the `grade` value to the `total` and increments the `gradeCounter`. Then lines 24–25 in the loop body input the next value from the user. Next, program control reaches the closing right brace of the loop at line 26, so execution continues with the test of the `while`'s condition (line 19). The condition uses the most recent `grade` entered by the user to determine whether the loop body should execute again.

The next `grade` is always input from the user immediately before the `while` condition is tested. This allows the program to determine whether the value just input is the sentinel value before processing that value (i.e., adding it to the `total`). If the sentinel value is input, the loop terminates, and the program does not add –1 to the `total`.

After the loop terminates, the `if`...`else` statement at lines 30–42 executes. The condition at line 30 determines whether any grades were input. If none were input, the `if`...`else` statement's `else` part executes and displays the message `"No grades were entered"`. After the `if`...`else` executes, the program terminates.

3.7.2 Converting Between Fundamental Types Explicitly and Implicitly

If at least one grade was entered, line 32 of Fig. 3.2

```
double average{static_cast<double>(total) / gradeCounter};
```

calculates the average. Recall from Fig. 3.1 that integer division yields an integer result. Even though `average` is declared as a `double`, if we had written line 32 as

```
double average{total / gradeCounter};
```

it would lose the fractional part of the quotient before the result of the division was used to initialize `average`.

`static_cast` Operator

To perform a floating-point calculation with integers in this example, you first create temporary floating-point values using the **`static_cast` operator**. Line 32 converts a temporary copy of its

operand in parentheses (`total`) to the type in angle brackets (`double`). The value stored in the `int` variable `total` is still an integer. Using a cast operator in this manner is called **explicit conversion**. `static_cast` is one of several cast operators we'll discuss.

Promotions

After the cast, the calculation consists of the temporary `double` copy of `total` divided by the `int` `gradeCounter`. For arithmetic, the compiler knows how to evaluate only expressions in which all the operand types are identical. To ensure this, the compiler performs an operation called **promotion** (also called **implicit conversion**) on selected operands. In an expression containing values of data types `int` and `double`, C++ **promotes** `int` operands to `double` values. So in line 32, C++ promotes a temporary copy of `gradeCounter`'s value to type `double`, then performs the division. Finally, `average` is initialized with the floating-point result. Section 5.6 discusses the allowed fundamental-type promotions.

Cast Operators for Any Type

Cast operators are available for use with every fundamental type and for other types, as you'll see beginning in Chapter 9. Simply specify the type in the angle brackets (`<` and `>`) that follow the `static_cast` keyword. It's a **unary operator**—that is, it has only one operand. Other unary operators include the unary plus (`+`) and minus (`-`) operators for expressions such as `-7` or `+5`. Cast operators have the second-highest precedence.

3.7.3 Formatting Floating-Point Numbers

Figure 3.2's formatting features are introduced here briefly. Online Chapter 19 explains them in depth.

`setprecision` Parameterized Stream Manipulator

Line 37's call to **`setprecision`**—`setprecision(2)`—indicates that floating-point values should be output with *two* digits of **precision** to the right of the decimal point (e.g., 92.37). `setprecision` is a **parameterized stream manipulator** because it requires an argument (in this case, `2`) to perform its task. Programs that use parameterized stream manipulators must include the header **`<iomanip>`** (line 4).

`fixed` Nonparameterized Stream Manipulator

The **non-parameterized stream manipulator** **`fixed`** (line 37) does not require an argument and indicates that floating-point values should be output in **fixed-point format**. This is as opposed to **scientific notation**[4], which displays a number between 1.0 and 10.0, multiplied by a power of 10. So, in scientific notation, the value 3,100.0 is displayed as `3.1e+03` (that is, 3.1×10^3). This format is useful for displaying very large or very small values.

---

4. Formatting using scientific notation is discussed further in online Chapter 19.

Fixed-point formatting forces a floating-point number to display without scientific notation. Fixed-point formatting also forces the decimal point and trailing zeros to print, even if the value is a whole-

number amount, such as 88.00. Without the fixed-point formatting option, 88.00 prints as 88 without the trailing zeros and decimal point.

The stream manipulators `setprecision` and `fixed` perform **sticky settings**. Once they're specified, all floating-point values formatted in your program will use those settings until you change them. Online Chapter 19 shows how to capture the stream format settings before applying sticky settings, so you can restore the original format settings later.

Rounding Floating-Point Numbers

When the stream manipulators `fixed` and `setprecision` are used, the printed value is **rounded** to the number of decimal positions specified by the current precision. The value in memory remains unaltered. For a precision of `2`, the values 87.946 and 67.543 are rounded to 87.95 and 67.54, respectively.[5]

5. In Fig. 3.2, if you do not specify `setprecision` and `fixed`, C++ uses four digits of precision by default. If you specify only `fixed`, C++ uses six digits of precision.

Together, lines 37 and 38 of Fig. 3.2 output the class average rounded to the nearest hundredth and with exactly two digits to the right of the decimal point. The three grades entered during the program's execution in Fig. 3.2 total 257, which yields the average 85.666... and displays the rounded value 85.67.

3.8 Nested Control Statements

We've seen that control statements can be stacked on top of one another (in sequence). In this case study, we examine the only other structured way control statements can be connected—**nesting** one control statement within another.

3.8.1 Problem Statement

Consider the following problem statement:

A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the exam. The college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is written a 1 if the student passed the exam or a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. Input each test result (i.e., a 1 or a 2). Display the message "Enter result" on the screen each time the program requests another test result.
2. Count the number of test results of each type.
3. Display a summary of the test results, indicating the number of students who passed and the number who failed.
4. If more than eight students passed the exam, print "Bonus to instructor!"

3.8.2 Implementing the Program

Figure 3.3 implements the program with counter-controlled iteration and shows two sample executions. Lines 8–10 and 16 declare the variables that are used to process the examination results. The `while` statement (lines 13–29) loops 10 times. During each iteration, the loop inputs and processes one exam result. Notice that the `if`...`else` statement (lines 20–25) for processing each result is nested in the `while` statement. If the `result` is `1`, the `if`...`else` statement increments `passes`; otherwise, it assumes the `result` is `2` and increments `failures`.[6] Line 28 increments `studentCounter` before the loop condition is tested again at line 13.

6. Assuming `result` is `2` could be a bad assumption if invalid data is entered. We'll discuss data-validation techniques later.

```
1   // fig03_03.cpp
2   // Analysis of examination results using nested control statements.
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      // initializing variables in declarations
8      int passes{0};
9      int failures{0};
10     int studentCounter{1};
11
12     // process 10 students using counter-controlled loop
13     while (studentCounter <= 10) {
14        // prompt user for input and obtain value from user
15        cout << "Enter result (1 = pass, 2 = fail): ";
16        int result;
17        cin >> result;
18
19        // if...else is nested in the while statement
20        if (result == 1) {
21           passes = passes + 1;
22        }
23        else {
24           failures = failures + 1;
25        }
26
27        // increment studentCounter so loop eventually terminates
28        studentCounter = studentCounter + 1;
29     }
30
31     // termination phase; prepare and display results
32     cout << "Passed: " << passes << "\nFailed: " << failures << "\n";
33
34     // determine whether more than 8 students passed
35     if (passes > 8) {
36        cout << "Bonus to instructor!\n";
37     }
38  }
```

```
Enter result (1 = pass, 2 = fail): 1
```

```
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 9
Failed: 1
Bonus to instructor!
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed: 6
Failed: 4
```

Fig. 3.3 Analysis of examination results using nested control statements.

After 10 values have been input, the loop terminates, and line 32 displays the number of `passes` and `failures`. The `if` statement at lines 35–37 determines whether more than eight students passed the exam and, if so, outputs the message `"Bonus to instructor!"`

Figure 3.3 shows the input and output from two sample executions. During the first, the condition at line 35 is true—more than eight students passed the exam, so the program outputs a message to give the instructor a bonus.

3.8.3 Preventing Narrowing Conversions with Braced Initialization

11 Consider the C++11 braced initialization in line 10 of Fig. 3.3:

```
int studentCounter{1};
```

Prior to C++11, you would have written this as

```
int studentCounter = 1;
```

For fundamental-type variables, braced initializers prevent **narrowing conversions** that could result in *data loss*. For example, the declaration

```
int x = 12.7;
```

attempts to assign the `double` value `12.7` to the `int` variable `x`. Here, C++ converts the `double` value to an `int` by truncating the floating-point part (`.7`). This is a narrowing conversion that loses data. So, this declaration assigns `12` to `x`. Compilers typically issue a warning for this but still compile the code.

However, using braced initialization, as in

```
int x{12.7};
```

Err yields a *compilation error*, helping you avoid a potentially subtle logic error. If you specify a whole-number `double` value, like `12.0`, you'll still get a compilation error. The initializer's type (`double`), not its value (`12.0`), determines whether a compilation error occurs.

The C++ standard document does not specify error-message wording. For the preceding declaration, Apple's Xcode compiler gives the error

```
Type 'double' cannot be narrowed to 'int' in initializer list
```

Visual Studio gives the error

```
conversion from 'double' to 'int' requires a narrowing conversion
```

and GNU C++ gives the error

```
type 'double' cannot be narrowed to 'int' in initializer list

[-Wc++11-narrowing]
```

We'll discuss additional braced-initializer features in later chapters.

A Look Back at Fig. 3.1

You might think that the following statement from Fig. 3.1

```
int average{total / 10}; // int division yields int result
```

Err contains a narrowing conversion, but `total` and `10` are both `int` values, so the initializer value is an `int`. If `total` were a `double` in the preceding statement or we used the `double` literal `10.0` for the denominator, then the initializer value would have type `double`, and the compiler would issue an error message for a narrowing conversion.

3.9 Compound Assignment Operators

You can abbreviate the statement

```
c = c + 3;
```

with the **addition compound assignment operator**, `+=`, as

```
c += 3;
```

The `+=` operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on the left. Thus, the assignment expression `c+=3` adds `3` to `c`. The following table shows all the arithmetic compound assignment operators, sample expressions and explanations of what the operators do:

Operator	Sample expression	Explanation	Assigns
	Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`		
`+=`	`c += 7`	`c = c + 7`	`10 to c`
`-=`	`d -= 4`	`d = d - 4`	`1 to d`
`*=`	`e *= 5`	`e = e * 5`	`20 to e`
`/=`	`f /= 3`	`f = f / 3`	`2 to f`
`%=`	`g %= 9`	`g = g % 9`	`3 to g`

3.10 Increment and Decrement Operators

The following table summarizes C++'s two unary operators for adding 1 to or subtracting 1 from the value of a numeric variable—these are the unary **increment operator**, `++`, and the unary **decrement operator**, `--`:

Operator	Operator name	Sample expression	Explanation
`++`	prefix increment	`++number`	Increment `number` by `1`, then use the new value of `number` in the expression in which `number` resides.
`++`	postfix increment	`number++`	Use the current value of `number` in the expression in which `number` resides, then increment `number` by `1`.
`--`	prefix decrement	`--number`	Decrement `number` by `1`, then use the new value of `number` in the expression in which `number` resides.

Operator	Operator name	Sample expression	Explanation
`--`	postfix decrement	`number--`	Use the current value of `number` in the expression in which `number` resides, then decrement `number` by `1`.

An increment or decrement operator that's prefixed to (placed before) a variable is called the **prefix increment** or **prefix decrement operator**, respectively. An increment or decrement operator that's postfixed to (placed after) a variable is called the **postfix increment** or **postfix decrement operator**, respectively.

Using the prefix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable **preincrements** (or **predecrements**) the variable. The variable is incremented (or decremented) by 1, then its new value is used in the expression in which it appears.

Using the postfix increment (or decrement) operator to add 1 to (or subtract 1 from) a variable **postincrements** (or **postdecrements**) the variable. The variable's current value is used in the expression in which it appears, then its value is incremented (or decremented) by 1. Unlike binary operators, the unary increment and decrement operators by convention should be placed next to their operands, with no intervening spaces.

Prefix Increment vs. Postfix Increment

Figure 3.4 demonstrates the difference between the prefix increment and postfix increment versions of the `++` increment operator. The decrement operator (`--`) works similarly.

```
1   // fig03_04.cpp
2   // Prefix increment and postfix increment operators.
3   #include <iostream>
4   using namespace std;
5
6   int main() {
7      // demonstrate postfix increment operator
8      int c{5};
9      cout << "c before postincrement: " << c << "\n"; // prints 5
10     cout << " postincrementing c: " << c++ << "\n"; // prints 5
11     cout << " c after postincrement: " << c << "\n"; // prints 6
12
13     cout << "\n"; // skip a line
14
15     // demonstrate prefix increment operator
16     c = 5;
17     cout << " c before preincrement: " << c << "\n"; // prints 5
18     cout << " preincrementing c: " << ++c << "\n"; // prints 6
19     cout << " c after preincrement: " << c << "\n"; // prints 6
20  }
```

```
c before postincrement: 5
   postincrementing c: 5
```

```
c after postincrement: 6
c before preincrement: 5
    preincrementing c: 6
 c after preincrement: 6
```

Fig. 3.4 Prefix increment and postfix increment operators.

Line 8 initializes the variable `c` to `5`, and line 9 outputs `c`'s initial value. Line 10 outputs the value of the expression `c++`. This expression postincrements the variable `c`, so `c`'s original value (`5`) is output, then `c`'s value is incremented (to 6). Thus, line 10 outputs `c`'s initial value (`5`) again. Line 11 outputs `c`'s new value (`6`) to prove that the variable's value was incremented in line 10. Line 16 resets `c`'s value to `5`, and line 17 outputs `c`'s value. Line 18 outputs the value of the expression `++c`. This expression preincrements `c`, so its value is incremented. Then the new value (`6`) is output. Line 19 outputs `c`'s value again to show that `c` is still `6` after line 18 executes.

Simplifying Statements with the Arithmetic Compound Assignment, Increment and Decrement Operators

The arithmetic compound assignment operators and the increment and decrement operators can be used to simplify program statements. For example, the three assignment statements in Fig. 3.3 (lines 21, 24 and 28)

```
passes = passes + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;
```

can be written more concisely with compound assignment operators as

```
passes += 1;
failures += 1;
studentCounter += 1;
```

with prefix increment operators as

```
++passes;
++failures;
++studentCounter;
```

or with postfix increment operators as

```
passes++;
failures++;
studentCounter++;
```

When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the same effect, and the prefix decrement and postfix decrement forms have the same effect. Only when a variable appears in the context of a larger expression does preincrementing or postincrementing the variable have a different effect (and similarly for predecrementing or postdecrementing).

Err Attempting to use the increment or decrement operator on an expression other than one to which a value can be assigned is a compilation error. For example, writing `++(x + 1)` is a syntax error, because `(x + 1)` is not a variable.

Operator Precedence and Grouping

The following table shows the precedence and grouping of the operators introduced to this point. The operators are shown top-to-bottom in decreasing order of precedence. The second column indicates the grouping of the operators at each level of precedence. Notice that the conditional operator (`?:`), the unary operators preincrement (`++`), predecrement (`--`), plus (`+`) and minus (`-`), and the assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=` group *right-to-left*. All other operators in this table group *left-to-right*.

Operators	Grouping
`++ -- static_cast<type>()`	left to right
`++ -- + -`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`?:`	right to left
`= += -= *= /= %=`	right to left

3.11 Fundamental Types Are Not Portable

You can view the complete list of C++ fundamental types and their typical ranges at

```
https://en.cppreference.com/w/cpp/language/types
```

In C and C++, an `int` on one machine might be represented by 16 bits (2 bytes) of memory, on a second machine by 32 bits (4 bytes), and on another machine by 64 bits (8 bytes). For this reason, code using integers is not always portable across platforms. You could write multiple versions of your programs to use different integer types on different platforms. Or you could use techniques to achieve various levels of portability.[7] In the next section, we'll show one way to achieve portability.

7. The integer types in the header `<cstdint>` (https://en.cppreference.com/w/cpp/types/integer) can be used to ensure that integer variables are correctly sized for your application across platforms.

Perf Among C++'s integer types are `int`, `long` and `long long`. The C++ standard requires type `int` to be at least 16 bits, type `long` to be at least 32 bits and type `long long` to be at least 64 bits. The standard also requires that an `int`'s size be less than or equal to a `long`'s size and that a `long`'s size be less than or equal to a `long long`'s size. Such "squishy" requirements create portability challenges but allow compiler implementers to optimize performance by matching fundamental types sizes to your machine's hardware.

3.12 Objects Natural Case Study: Arbitrary-Sized Integers

The range of values an integer type supports depends on the number of bytes used to represent the type on a particular computer. For example, a four-byte `int` can store 2^{32} possible values in the range –2,147,483,648 to 2,147,483,647. On most systems, a `long long` integer is 8 bytes and can store 2^{64} possible values in the range –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Some Applications Need Numbers Outside a `long long` Integer's Range

Consider factorial calculations. A factorial is the product of the integers from 1 to a given value. The factorial of 5 (written 5!) is 1 * 2 * 3 * 4 * 5, which is 120. The highest factorial value we can represent in a 64-bit integer is 20!, which is 2,432,902,008,176,640,000. Factorials quickly grow outside the range representable by a `long long` integer. With big data getting bigger quickly, an increasing number of real-world applications will exceed the limitations of `long long` integers.

Sec Another application requiring huge integers is cryptography—an essential aspect of securing data transmitted between computers over the Internet. Many cryptography algorithms perform calculations using 128-bit or 256-bit integer values—far larger than we can represent with C++'s fundamental types.

Arbitrary-Precision Integers with Class `BigNumber`

Any application requiring integers outside `long long`'s range requires special processing. Unfortunately, the C++ standard library does not (yet) have a class for arbitrary-precision integers. So, for this example, we'll dive into the vast world of open-source class libraries to demonstrate one of the many C++ classes you can use to create and manipulate arbitrary-precision integers. We'll use the class `BigNumber` from

```
https://github.com/limeoats/BigNumber
```

For your convenience, we included the download in the `examples` folder's `libraries` folder. Be sure to read the open-source license terms included in the provided `LICENSE.md` file.

To use `BigNumber`, you don't have to understand how it's implemented.[8] You simply include its header file (`bignumber.h`), easily create objects of the class, then use them in your code. Figure 3.5 demonstrates `BigNumber` and shows a sample output. For this example, we'll use the maximum `long long` integer value to show that we can create an even bigger integer with `BigNumber`. At the end of this section, we show how to compile and run the code on our preferred compilers.

8. After you get deeper into C++, you might want to peek at `BigNumber`'s source code (approximately 1,000 lines) to see how it's implemented. In our object-oriented programming presentation later in this book, you'll learn a variety of techniques that you can use to create your own big-integer class.

```cpp
// fig03_05.cpp
// Integer ranges and arbitrary-precision integers.
#include <iostream>
#include "bignumber.h"
using namespace std;

int main() {
   // use the maximum long long fundamental type value in calculations
   const long long value1{9'223'372'036'854'775'807LL}; // long long max
   cout << "long long value1: " << value1
      << "\nvalue1 - 1 = " << value1 - 1 // OK
      << "\nvalue1 + 1 = " << value1 + 1; // result is undefined

   // use an arbitrary-precision integer
   const BigNumber value2{value1};
   cout << "\n\nBigNumber value2: " << value2
      << "\nvalue2 - 1 = " << value2 - 1 // OK
      << "\nvalue2 + 1 = " << value2 + 1; // OK

   // powers of 100,000,000 with long long
   long long value3{100'000'000};
   cout << "\n\nvalue3: " << value3;

   int counter{2};

   while (counter <= 5) {
      value3 *= 100'000'000; // quickly exceeds maximum long long value
      cout << "\nvalue3 to the power " << counter << ": " << value3;
      ++counter;
   }

   // powers of 100,000,000 with BigNumber
   BigNumber value4{100'000'000};
   cout << "\n\nvalue4: " << value4 << "\n";

   counter = 2;

   while (counter <= 5) {
      cout << "value4.pow(" << counter << "): "
         << value4.pow(counter) << "\n";
      ++counter;
   }

   cout << "\n";
}
```

```
long long value1: 9223372036854775807
value1 - 1: 9223372036854775806                    OK
value1 + 1: -9223372036854775808                   Incorrect result

BigNumber value2: 9223372036854775807
value2 - 1: 9223372036854775806                    OK
value2 + 1: 9223372036854775808                    OK
value3: 100000000
value3 to the power 2: 10000000000000000           OK
value3 to the power 3: 2003764205206896640         Incorrect result
```

```
value3 to the power 4: -8814407033341083648                Incorrect result
value3 to the power 5: -5047021154770878464                Incorrect result

value4: 100000000
value4.pow(2): 10000000000000000                           OK
value4.pow(3): 1000000000000000000000000                   OK
value4.pow(4): 100000000000000000000000000000000           OK
value4.pow(5): 10000000000000000000000000000000000000000   OK
```

Fig. 3.5 Integer ranges and arbitrary-precision integers.

Including a Header That Is Not in the C++ Standard Library

In an `#include` directive, headers that are located in the same folder as the application or in one of the application's subfolders typically are placed in double quotes (`""`) rather than angle brackets (`<>`). The double quotes tell the compiler the header is in your application's folder or another folder that you specify.

What Happens When You Exceed the Maximum `long long` Integer Value?

Line 9 initializes the variable `value1` with the maximum `long long` value on our system:[9]

9. The platforms on which we tested this book's code each have as their maximum `long long` integer value 9,223,372,036,854,775,807. You can determine this value programmatically with the expression `std::numeric_limits<long long>::max()`, which uses class `numeric_limits` from the C++ standard library header `<limits>`. The `<>` and `::` notations used in this expression are covered in later chapters, so we used the literal value 9,223,372,036,854,775,807 in Fig. 3.5.

```
long long value1{9'223'372'036'854'775'807LL}; // max long long value
```

14 Typing numeric literals with many digits can be error-prone. To make such literals more readable and reduce errors, C++14 introduced the **digit separator** `'` (a single-quote character), which you insert between groups of digits in numeric literals—we used it to separate groups of three digits. Also, note the `LL` ("el el") at the end of the literal value—this indicates that the literal is a `long long` integer.

Line 10 displays `value1`, then line 11 subtracts one from it to demonstrate a valid calculation. Next, we attempt to add `1` to `value1`, which already contains the maximum `long long` value. All our compilers displayed as the result the *minimum `long long` value*. The C++ standard actually says the result of this calculation is **undefined behavior**. Such behaviors can differ between systems—ours displayed an incorrect value, but other systems could terminate the program and display an error message. This is another example of why the fundamental integer types are not portable.

Performing the Same Operations with a `BigNumber` Object

Lines 15–18 use a `BigNumber` object to repeat the operations from lines 9–12. We create a `BigNumber` object named `value2` and initialize it with `value1`, which contains the maximum value of a `long long` integer:

```
BigNumber value2{value1};
```

Next, we display the `BigNumber`, then subtract one from it and display the result. Line 18 adds one to `value2`, which contains the maximum value of a `long long`. `BigNumber` handles arbitrary-precision

integers, so it *correctly performs this calculation*. The result is a value that C++'s fundamental integer types cannot handle on our systems.

`BigNumber` supports all the typical arithmetic operations, including `+` and `-` used in this program. The compiler already knows how to use arithmetic operators with fundamental numeric types, but it has to be taught how to handle those operators for class objects. We discuss that process—called operator overloading—in Chapter 11.

Powers of 100,000,000 with `long long` Integers

Lines 21–30 calculate powers of 100,000,000 using `long long` integers. First, we create the variable `value3` and display its value. Lines 26–30 loop four times. The calculation

```
value3 *= 100'000'000; // quickly exceeds maximum long long value
```

multiples `value3`'s current value by 100,000,000 to raise `value3` to the next power. As you can see in the program's output, only the loop's first iteration produces a correct result.

Powers of 100,000,000 with `BigNumber` Objects

To demonstrate that `BigNumber` can handle significantly larger values than the fundamental integer types, lines 33–42 calculate powers of 100,000,000. First, we create `BigNumber value4` and display its initial value. Lines 38–42 loop four times. The calculation

```
value4.pow(counter)
```

calls `BigNumber` member function **`pow`** to raise `value4` to the power `counter`.[10] `BigNumber` correctly handles each calculation, producing massive values far outside the ranges supported by our Windows, macOS and Linux systems' fundamental integer types.

10. We chose to demonstrate `BigNumber`'s `pow` member function here, but we also could have used `*=`, as we did for the `long long` variable in line 27.

Perf Though a `BigNumber` can represent any integer value, it does not match your system's hardware. So you'll sacrifice performance in exchange for the flexibility `BigNumber` provides.

Compiling and Running the Example in Microsoft Visual Studio

In Microsoft Visual Studio:

1. Create a new project, as described in Section 1.2.
2. In the **Solution Explorer**, right-click the project's **Source Files** folder and select **Add > Existing Item...**.
3. Navigate to the `fig03_05` folder, select `fig03_05.cpp` and click **Add**.
4. Repeat Steps 2–3 for `bignumber.cpp` from `examples\libraries\BigNumber\src`.
5. In the **Solution Explorer**, right-click the project's name and select **Properties...**.
6. Under **Configuration Properties**, select **C/C++**. Then, on the right side of the dialog, add to the **Additional Include Directories** the full path to the `BigNumber\src` folder on your system. For our system, this was

```
C:\Users\account\Documents\examples\libraries\BigNumber\src
```

7. Click **OK**.
8. Type *Ctrl + F5* to compile and run the program.[11]

11. Visual C++ may issue warnings for code in class `BigNumber`.

Compiling and Running the Example in GNU g++

For GNU g++ (these instructions also work from a Terminal window on macOS):

1. At your command line, change to this example's `fig03_05` folder.
2. To compile the program, type the following command (on one line)—the `-I` option specifies additional folders in which the compiler should search for header files:

```
g++ -std=c++2a -I ../../libraries/BigNumber/src fig03_05.cpp
   ../../libraries/BigNumber/src/bignumber.cpp -o fig03_05
```

3. Type the following command to execute the program:

```
./fig03_05
```

Compiling and Running the Example in Apple Xcode

In Apple Xcode:

1. Create a new project, as described in Section 1.2, and delete `main.cpp`.
2. Drag `fig03_05.cpp` from the `fig03_05` folder in the Finder onto your project's source-code folder in Xcode, then click **Finish** in the dialog that appears.
3. Drag `bignumber.h` and `bignumber.cpp` from the folder `examples/libraries/ BigNumber/src` onto your project's source-code folder in Xcode, then click **Finish** in the dialog that appears.
4. Type ⌘ + *R* to compile and run the program.

3.13 C++20: Text Formatting with Function `format`

C++20 introduces powerful new string-formatting capabilities via the `format` function (in header `<format>`). These capabilities greatly simplify C++ formatting by using a syntax similar to formatting in Python, Microsoft's .NET languages (like C# and Visual Basic) and the newer language Rust.[12] You'll see that the C++20 text-formatting capabilities are more concise and more powerful than those in earlier C++ versions. We'll primarily use these new text-formatting capabilities going forward. Online Chapter 19 presents old-style formatting details in case you work with legacy software that uses the old style.

12. Victor Zverovich, "Text Formatting," July 16, 2019. Accessed November 11, 2021. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html.

C++20 String Formatting Is Not Yet Implemented

At the time of this writing (December 2021), only Visual C++ had implemented C++20's new text-formatting capabilities. The open-source `{fmt}` library at

```
https://github.com/fmtlib/fmt
```

provides a full implementation of the new features.[13,14] We'll use this library until the C++20 compilers implement text formatting.[15] For your convenience, we have included the complete `{fmt}` library download in the `examples` folder's `libraries` subfolder. See the compilation steps after the example. The library's license terms are available at

13. According to http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0645r10.html, which is the C++ standards committee proposal for C++20 text formatting.

14. C++20's text-formatting features are a subset of the features provided by the `{fmt}` library.
15. Some of our C++20 Feature Mock-Up sections present code that does not compile or run. Once the compilers implement those features, we'll retest the code, update our digital products and post updates for our print products at https://deitel.com/c-plus-plus-20-for-programmers. The code in this example runs, but uses the `{fmt}` open-source library to demonstrate features that C++20 compilers will eventually support.

```
https://github.com/fmtlib/fmt/blob/master/LICENSE.rst
```

Format String Placeholders

The `format` function's first argument is a **format string** containing **placeholders** delimited by curly braces (`{` and `}`). The function replaces the placeholders with the values of the function's other arguments, as demonstrated in Fig. 3.6.

```
1  // fig03_06.cpp
2  // C++20 string formatting.
3  #include <iostream>
4  #include <fmt/format.h> // C++20: This will be #include <format>
5  using namespace std;
6  using namespace fmt; // not needed in C++20
7
8  int main() {
9     string student{"Paul"};
10    int grade{87};
11
12    cout << format("{}'s grade is {}\n", student, grade);
13 }
```

```
Paul's grade is 87
```

Fig. 3.6 C++20 string formatting.

Placeholders Are Replaced Left-to-Right

The `format` function replaces its format string argument's placeholders left-to-right by default. So, line 12's `format` call inserts into the format string

```
"{}'s grade is {}"
```

`student`'s value (`"Paul"`) in the first placeholder and `grade`'s value (`87`) in the second placeholder, then returns the string

```
"Paul's grade is 87"
```

Compiling and Running the Example in Microsoft Visual Studio

The steps below are the same as those in Section 3.12 with the following changes:

- In *Step 3*, navigate to the `fig03_06` folder and add `fig03_06.cpp` to your project's **Source Files** folder.
- In *Step 4*, navigate to the `examples\libraries\fmt\src` folder and add the file `format.cc` to your project's **Source Files** folder.
- In *Step 6*, add to the **Additional Include Directories** the full path to the `{fmt}` library's `include` folder on your system. For our system, this was

```
C:\Users\account\Documents\examples\libraries\fmt\include
```

Compiling and Running the Example in GNU g++

For GNU g++ (these instructions also work from a Terminal window on macOS):

1. At your command line, change folders to this example's `fig03_06` folder.
2. Type the following command on one line to compile the program:

```
g++ -std=c++2a -I ../../libraries/fmt/include fig03_06.cpp
   ../../libraries/fmt/src/format.cc -o fig03_06
```

3. Type the following command to execute the program:

```
./fig03_06
```

Compiling and Running the Example in Apple Xcode

The steps for this example are the same as those in Section 3.12 with one change: In *Step 2*, drag the files `fig03_06.cpp`, `format.cc` and the folder `fmt` from the `fig03_06` folder onto your project's source-code folder in Xcode.

3.14 Wrap-Up

Only three types of control statements—sequence, selection and iteration—are needed to develop any algorithm. We demonstrated the `if` single-selection statement, the `if`...`else` double-selection statement and the `while` iteration statement. We used control-statement stacking to total and compute the average of a set of student grades with counter- and sentinel-controlled iteration. We used control-statement nesting to analyze and make decisions based on a set of exam results. We introduced the compound assignment operators and the increment and decrement operators.

We discussed why C++'s fundamental types are not guaranteed to be the same size across platforms, then used objects of the open-source class `BigNumber` to perform integer arithmetic with values outside the range supported by our systems. Finally, we introduced C++20's new text formatting in the context of the open-source `{fmt}` library. Once your preferred compiler fully supports the C++20 `<format>` header, you'll no longer need the separate `{fmt}` library. Chapter 4 continues our control-statements discussion, introducing the `for`, `do`...`while` and `switch` statements. It also introduces the logical operators for creating compound conditions.

4. Control Statements, Part 2

Objectives

In this chapter, you'll:

- Use the `for` and `do...while` iteration statements.
- Perform multiple selection using the `switch` selection statement.
- Use C++17's `[[fallthrough]]` attribute in `switch` statements.
- Use C++17's selection statements with initializers.
- Use the `break` and `continue` statements to alter the flow of control.
- Use the logical operators to form compound conditions in control statements.
- Understand the representational errors associated with using floating-point data to hold monetary values.
- Continue our Objects-Natural approach with a case study that uses an open-source ZIP compression/decompression library to create and read ZIP files.
- Use more C++20 text-formatting capabilities.

Outline

4.1 Introduction
4.2 Essentials of Counter-Controlled Iteration
4.3 `for` Iteration Statement
4.4 Examples Using the `for` Statement
4.5 Application: Summing Even Integers
4.6 Application: Compound-Interest Calculations
4.7 `do...while` Iteration Statement
4.8 `switch` Multiple-Selection Statement
4.9 C++17 Selection Statements with Initializers
4.10 `break` and `continue` Statements
4.11 Logical Operators
4.11.1 Logical AND (`&&`) Operator
4.11.2 Logical OR (`||`) Operator
4.11.3 Short-Circuit Evaluation
4.11.4 Logical Negation (`!`) Operator
4.11.5 Example: Producing Logical-Operator Truth Tables
4.12 Confusing the Equality (`==`) and Assignment (`=`) Operators
4.13 Objects-Natural Case Study: Using the `miniz-cpp` Library to Write and Read ZIP files
4.14 C++20 Text Formatting with Field Widths and Precisions
4.15 Wrap-Up

4.1 Introduction

17 20 This chapter introduces the `for`, `do...while`, `switch`, `break` and `continue` control statements. We explore the essentials of counter-controlled iteration. We use compound-interest calculations to begin investigating the issues of processing monetary amounts. First, we discuss the representational errors associated with floating-point types. We use a `switch` statement to count the number of A, B, C,

D and F grade equivalents in a set of numeric grades. We show C++17's enhancements that allow you to initialize one or more variables of the same type in the headers of `if` and `switch` statements. We discuss the logical operators, which enable you to combine simple conditions to form compound conditions. In our Objects-Natural case study, we continue using objects of preexisting classes with the miniz-cpp open-source library for creating and reading compressed ZIP archive files. Finally, we introduce more of C++20's powerful and expressive text-formatting features.

4.2 Essentials of Counter-Controlled Iteration

This section uses the `while` iteration statement introduced in Chapter 3 to formalize the elements of counter-controlled iteration:

1. a **control variable** (or loop counter)
2. the control variable's **initial value**
3. the control variable's **increment** that's applied during each iteration of the loop
4. the **loop-continuation condition** that determines if looping should continue.

Consider Fig. 4.1, which uses a loop to display the numbers from 1 through 10.

```
1  // fig04_01.cpp
2  // Counter-controlled iteration with the while iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int counter{1}; // declare and initialize control variable
8
9     while (counter <= 10) { // loop-continuation condition
10       cout << counter << " ";
11       ++counter; // increment control variable
12    }
13
14    cout << "\n";
15 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.1 Counter-controlled iteration with the `while` iteration statement.

In Fig. 4.1, lines 7, 9 and 11 define the elements of counter-controlled iteration. Line 7 declares the control variable (`counter`) as an `int`, reserves space for it in memory and sets its initial value to `1`. Declarations that require initialization are executable statements. Variable declarations that also reserve memory are **definitions**. We'll generally use the term "declaration," except when the distinction is important.

Line 10 displays `counter`'s value once per iteration of the loop. Line 11 increments the control variable by 1 for each iteration of the loop. The `while`'s loop-continuation condition (line 9) tests whether the value of the control variable is less than or equal to `10` (the final value for which the condition is true). The loop terminates when the control variable exceeds `10`.

Floating-point values are approximate, so controlling counting loops with floatingpoint variables can result in imprecise counter values and inaccurate termination tests, which can prevent a loop from terminating. For that reason, always control counting loops with integer variables.

4.3 `for` Iteration Statement

The **`for` iteration statement** specifies the counter-controlled-iteration details in a single line of code. Figure 4.2 reimplements the application of Fig. 4.1 using a `for` statement.

```
1  // fig04_02.cpp
2  // Counter-controlled iteration with the for iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     // for statement header includes initialization,
8     // loop-continuation condition and increment
9     for (int counter{1}; counter <= 10; ++counter) {
10       cout << counter << " ";
11    }
12
13    cout << "\n";
14 }
```

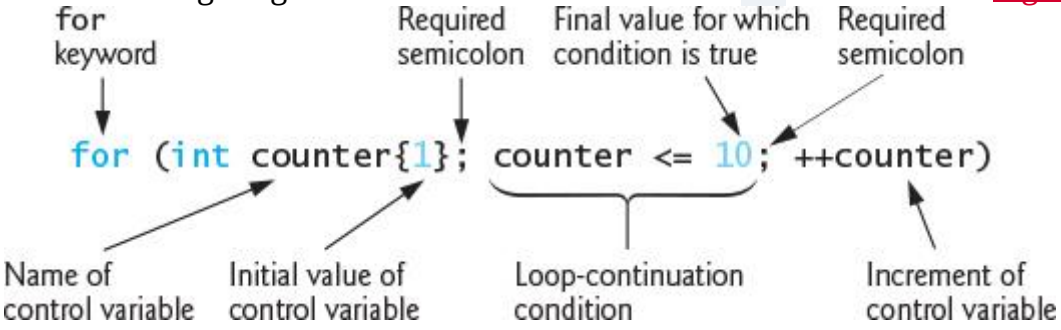
```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.2 Counter-controlled iteration with the `for` iteration statement.

When the `for` statement (lines 9–11) begins executing, the control variable `counter` is declared and initialized to `1`. Next, the program tests the loop-continuation condition between the two required semicolons (`counter <= 10`). Because `counter`'s initial value is `1`, the condition is true. So, line 10 displays `counter`'s value (`1`). After executing line 10, `++counter` to the right of the second semicolon increments `counter`. Then the program performs the loop-continuation test again to determine whether to proceed with the loop's next iteration. At this point, `counter`'s value is `2` and the condition is still true, so the program executes line 10 again. This process continues until the loop has displayed the numbers 1–10 and `counter`'s value becomes `11`. At this point, the loop-continuation test fails, iteration terminates and the program continues with the first statement after the loop (line 13).

A Closer Look at the `for` Statement's Header

The following diagram takes a closer look at the `for` statement in Fig. 4.2:



The first line—including the keyword `for` and everything in the parentheses after `for` (line 9 in Fig. 4.2)—is sometimes called the **`for` statement header**. The `for` header "does it all"—it specifies each item needed for counter-controlled iteration with a control variable.

General Format of a `for` Statement

The general format of the `for` statement is

```
for (initialization; loopContinuationCondition; increment) {
   statement
}
```
where

- *initialization* names the loop's control variable and provides its initial value,
- *loopContinuationCondition*—between the two required semicolons—determines whether the loop should continue executing, and
- *increment* modifies the control variable's value so that the loop-continuation condition eventually becomes false.

If the loop-continuation condition is initially false, the program does not execute the `for` statement's body. Instead, execution proceeds with the statement following the `for`.

Scope of a `for` Statement's Control Variable

If the *initialization* expression declares the control variable, it can be used only in that `for` statement—not beyond it. This restricted use is known as the variable's **scope**, which defines its lifetime and where it can be used in a program. For example, a variable's scope is from its declaration point to the right brace that closes the block. As you'll see in Chapter 5, it's good practice to define each variable in the smallest scope needed.

Expressions in a `for` Statement's Header Are Optional

All three expressions in a `for` header are optional. If you omit the *loopContinuationCondition*, the condition is always true, creating an infinite loop. You might omit the *initialization* expression if the program initializes the control variable before the loop. You might omit the *increment* expression if the program calculates the increment in the loop's body or if no increment is needed.

The increment expression in a `for` acts like a stand-alone statement at the end of the `for`'s body. Therefore, the increment expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

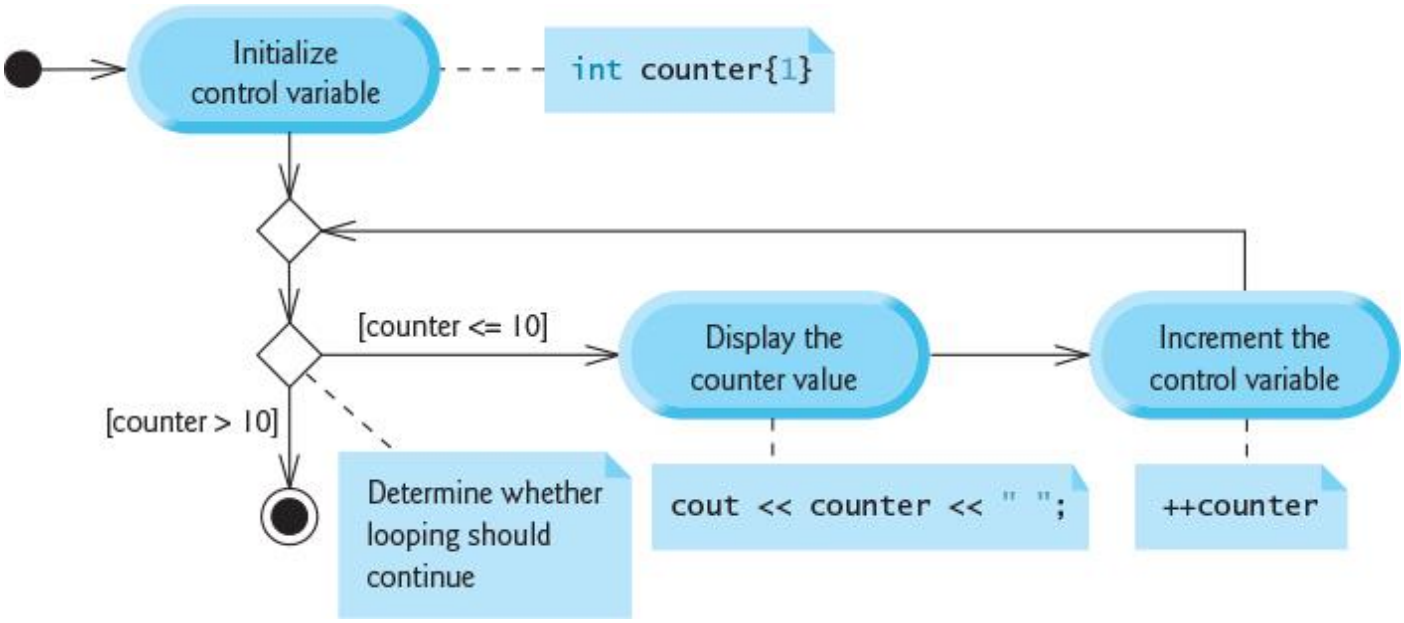
are equivalent in a `for` statement. In this case, the increment expression does not appear in a larger expression, so preincrementing and postincrementing have the same effect. We prefer preincrement. In Chapter 11's operator-overloading discussion, you'll see that preincrement can have a performance advantage.

Using a `for` Statement's Control Variable in the Statement's Body

Err Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is not required. The value of the control variable can be changed in a `for` loop's body, but doing so can lead to subtle errors. If a program must modify the control variable's value in the loop's body, prefer `while` to `for`.

UML Activity Diagram of the `for` Statement

Below is the UML activity diagram of the `for` statement in Fig. 4.2—it makes it clear that initialization occurs once, before the condition is tested the first time. Incrementing occurs after the body statement executes:



4.4 Examples Using the `for` Statement

The following examples show techniques for varying the control variable in a `for` statement. In each case, we write only the appropriate `for` header. Note the change in the relational operator for the loops that decrement the control variable.

Vary the control variable from `1` to `100` in increments of `1`.

```
for (int i{1}; i <= 100; ++i)
```

Vary the control variable from `100` *down* to `1` in *decrements* of `1`.

```
for (int i{100}; i >= 1; --i)
```

Vary the control variable from `7` to `77` in increments of `7`.

```
for (int i{7}; i <= 77; i += 7)
```

Vary the control variable from `20` *down* to `2` in *decrements* of `2`.

```
for (int i{20}; i >= 2; i -= 2)
```

Vary the control variable over the values `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for (int i{2}; i <= 20; i += 3)
```

Vary the control variable over the values `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for (int i{99}; i >= 0; i -= 11)
```

Do not use equality operators (`!=` or `==`) in a loop-continuation condition if the loop's control variable increments or decrements by more than `1`. For example, in the `for` statement header

```
for (int counter{1}; counter != 10; counter += 2)
```

`counter != 10` never becomes false (resulting in an infinite loop) because `counter` increments by 2 after each iteration, producing only the odd values (3, 5, 7, 9, 11, ...).

4.5 Application: Summing Even Integers

The application in Fig. 4.3 uses a `for` statement to sum the even integers from 2 to 20 and store the result in `int` variable `total`. Each iteration of the loop (lines 10–12) adds control variable `number`'s value to variable `total`.

```
1  // fig04_03.cpp
2  // Summing integers with the for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int total{0};
8
9     // total even integers from 2 through 20
10    for (int number{2}; number <= 20; number += 2) {
11       total += number;
12    }
13
14    cout << "Sum is " << total << "\n";
15 }
```

```
Sum is 110
```

Fig. 4.3 Summing integers with the `for` statement.

A `for` statement's initialization and increment expressions can be comma-separated lists containing multiple initialization expressions or multiple increment expressions. Although this is discouraged, you could merge the `for` statement's body (line 11) into the increment portion of the `for` header by using a comma operator as in

```
for (int number{2}; number <= 20; total += number, number += 2) { }
```

The comma between the expressions `total += number` and `number += 2` is the **comma operator**, which guarantees that a list of expressions evaluates from left to right. The comma operator has the lowest precedence of all C++ operators. The value and type of a comma-separated list of expressions is the value and type of the rightmost expression, respectively. The comma operator is often used in `for` statements that require multiple initialization expressions or multiple increment expressions.

4.6 Application: Compound-Interest Calculations

Let's compute compound interest with a `for` statement. Consider the following problem:

A person invests $1,000 in a savings account yielding 5% interest. Assuming all interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p\,(1 + r)^n$$

where

p is the original amount invested (i.e., the principal),
r is the annual interest rate (e.g., use 0.05 for 5%),
n is the number of years, and
a is the amount on deposit at the end of the *n*th year.

The solution (Fig. 4.4) uses a loop to perform the calculation for each of the 10 years the money remains on deposit. We use `double` values here for the monetary calculations. Then we discuss the problems with using floating-point types to represent monetary amounts. For financial applications that require precise monetary calculations and rounding control, consider using an open-source library such as `Boost.Multiprecision`.[1]

1. John Maddock and Christopher Kormanyos, "Chapter 1. Boost.Multiprecision." Accessed November 19, 2021. https://www.boost.org/doc/libs/master/libs/multiprecision/doc/html/index.html.

Lines 12–13 initialize `double` variable `principal` to `1000.00` and `double` variable `rate` to `0.05`. C++ treats floating-point literals like `1000.00` and `0.05` as type `double`. Similarly, C++ treats whole numbers like `7` and `-22` as type `int`.[2] Lines 15–16 display the initial principal and the interest rate.

2. Section 3.12 showed that C++'s integer types cannot represent all integer values. Choose the correct type for the range of values you need to represent. You may designate that an integer literal has type `long` or `long long` by appending `L` or `LL`, respectively, to the literal value.

```
1  // fig04_04.cpp
2  // Compound-interest calculations with for.
3  #include <iostream>
4  #include <iomanip>
5  #include <cmath> // for pow function
6  using namespace std;
7
8  int main() {
9     // set floating-point number format
10    cout << fixed << setprecision(2);
11
12    double principal{1000.00}; // initial amount before interest
13    double rate{0.05}; // interest rate
14
15    cout << "Initial principal: " << principal << "\n";
16    cout << " Interest rate: " << rate << "\n";
17
18    // display headers
19    cout << "\nYear" << setw(20) << "Amount on deposit" << "\n";
20
21    // calculate amount on deposit for each of ten years
22    for (int year{1}; year <= 10; ++year) {
23       // calculate amount on deposit at the end of the specified year
24       double amount{principal * pow(1.0 + rate, year)} ;
25
```

```
26          // display the year and the amount
27          cout << setw(4) << year << setw(20) << amount << "\n";
28       }
29    }
```

```
Initial principal: 1000.00
   Interest rate: 0.05

Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

Fig. 4.4 Compound-interest calculations with `for`.

Formatting with Field Widths and Justification

Line 10 before the loop and line 27 in the loop combine to print the `year` and `amount` values. We specify the formatting with the parameterized stream manipulators `setprecision` and **`setw`** and the nonparameterized stream manipulator `fixed`. The stream manipulator `setw(4)` specifies that the next value output should appear in a **field width** of 4—i.e., `cout <<` prints the value with at least four character positions. If the value to be output requires fewer than four character positions, the value is right-aligned in the field by default. If the value to be output has more than four character positions, C++ extends the field width to the right to accommodate the entire value. To left-align values, output non-parameterized stream manipulator **`left`** (found in header `<iostream>`). You can restore right-alignment by outputting nonparameterized stream manipulator **`right`**.

The other formatting in the output statements displays variable `amount` as a fixed-point value with a decimal point (`fixed` in line 10) right-aligned in a field of 20 character positions (`setw(20)` in line 27) and two digits of precision to the right of the decimal point (`setprecision(2)` in line 10). We applied the sticky stream manipulators `fixed` and `set-precision` to the output stream `cout` before the `for` loop because these format settings remain in effect until they're changed, and they do not need to be applied during each iteration of the loop. However, the field width specified with `setw` applies only to the next value output. Chapter 19 discusses `cin`'s and `cout`'s formatting capabilities in detail. We continue discussing C++20's powerful new text-formatting capabilities in Section 4.14.

Performing the Interest Calculations with Standard Library Function `pow`

The `for` statement (lines 22–28) iterates 10 times, varying the `int` control variable `year` from 1 to 10 in increments of 1. Variable `year` represents n in the problem statement.

C++ does not include an exponentiation operator, so we use the **standard library function `pow`** (line 24) from the header `<cmath>` (line 5). The call `pow(x, y)` calculates the value of x raised to the yth power. The function receives two `double` arguments and returns a `double` value. Line 24 performs the calculation $a = p(1 + r)^n$, where a is `amount`, p is `principal`, r is `rate` and n is `year`.

Perf Function `pow`'s first argument—the calculation `1.0 + rate`—produces the same result each time through the loop, so repeating it in every iteration of the loop is wasteful. To improve program performance, many of today's optimizing compilers place such calculations before loops in the compiled code.

Floating-Point Number Precision and Memory Requirements

A `float` represents a **single-precision floating-point number**. Most of today's systems store these in four bytes of memory with approximately seven significant digits. A `double` represents a **double-precision floating-point number**. Most of today's systems store these in eight bytes of memory with approximately 15 significant digits—approximately double the precision of `float`s. Most programmers use type `double`. C++ treats floating-point numbers such as 3.14159 in a program's source code as `double` values by default. Such values in the source code are known as **floating-point literals**.

The C++ standard requires only that type `double` provide at least as much precision as `float`. There is also type `long double`, which provides at least as much precision as `double`. For a complete list of C++ fundamental types and their typical ranges, see

```
https://en.cppreference.com/w/cpp/language/types
```

Floating-Point Numbers Are Approximations

Err In conventional arithmetic, floating-point numbers often arise as a result of division. Dividing 10 by 3, the result is 3.3333333..., with the sequence of 3s repeating infinitely. The computer allocates a fixed amount of space to hold such a value, so the stored value can be only an approximation. Floating-point types such as `double` suffer from what is referred to as **representational error**. Assuming that floating-point numbers are represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results.

Floating-point numbers have numerous applications, especially for measured values. For example, when we speak of a "normal" body temperature of 98.6 degrees Fahrenheit, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it actually might be 98.594732103. Calling this number 98.6 is fine for most body temperature calculations. Generally, `double` is preferred over `float`, because `double`s represent floating-point numbers more precisely.[3]

3. Nowadays, the standard floating-point representation is IEEE 754 (https://en.wikipedia.org/wiki/IEEE_754).

A Warning about Displaying Rounded Values

This example declared `double` variables `amount`, `principal` and `rate` to be of type `double`. Unfortunately, floating-point numbers can cause trouble with fractional dollar amounts. Here's a simple explanation of what can go wrong when floating-point numbers are used to represent dollar amounts that are displayed with two digits to the right of the decimal point. Two calculated dollar amounts stored in the machine could be 14.234 (rounded to 14.23 for display purposes) and 18.673 (rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would typically be rounded to 32.91 for display purposes. Thus, your output could appear as

```
  14.23
+ 18.67
  32.91
```

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

Even Common Dollar Amounts Can Have Floating-Point Representational Errors

Even simple dollar amounts can have representational errors when they're stored as `double`s. To see this, we created a simple program that defined the variable `d` as follows:

```
double d{123.02};
```

We displayed `d`'s value with 20 digits of precision to the right of the decimal point. The resulting output showed 123.02 as 123.0199999..., which is another example of a representational error. Though some dollar amounts can be represented precisely as `double`s, many cannot. This is a common problem in many programming languages. Later in the book, we create and use classes that handle monetary amounts precisely.

4.7 `do`... while Iteration Statement

In a `while` statement, the program tests the loop-continuation condition before executing the loop's body. If it's false, the body never executes. The **`do`...`while` iteration statement** tests the loop-continuation condition after executing the loop's body; so, the body always executes at least once. Figure 4.5 uses a `do`...`while` to output the numbers 1–10. Line 7 declares and initializes control variable `counter`. Upon entering the `do`...`while` statement, line 10 outputs `counter`'s value and line 11 increments `counter`. Then the program evaluates the loop-continuation test at the bottom of the loop (line 12). If the condition is true, the loop continues at the first body statement (line 10). If the condition is false, the loop terminates, and the program continues at the next statement after the loop.

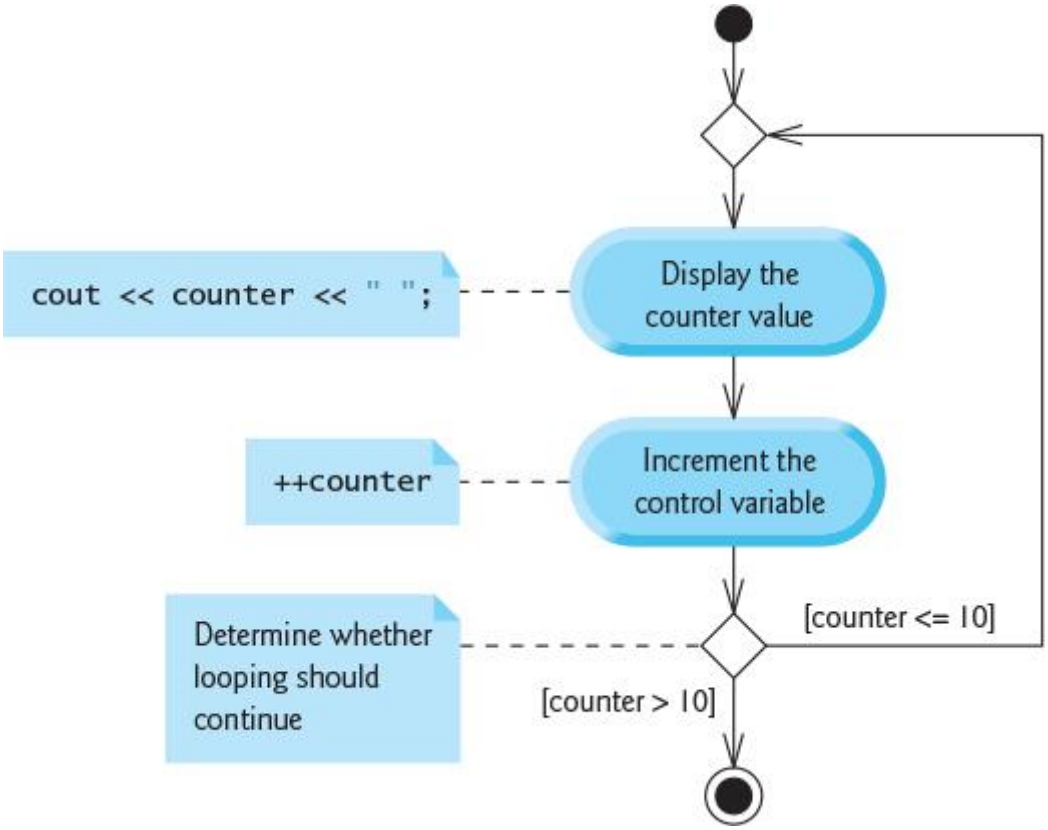
```
1  // fig04_05.cpp
2  // do...while iteration statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int counter{1};
8
9     do {
10       cout << counter << " ";
11       ++counter;
12    } while (counter <= 10); // end do...while
13
14    cout << "\n";
15 }
```

```
1 2 3 4 5 6 7 8 9 10
```

Fig. 4.5 | `do`...`while` iteration statement.

UML Activity Diagram for the `do`...`while` Iteration Statement

The `do`...`while`'s UML activity diagram makes it clear that the loop-continuation condition is not evaluated until after the loop performs the action state at least once:



4.8 `switch` Multiple-Selection Statement

C++ provides the **`switch` multiple-selection** statement to choose among many different actions based on the possible values of a variable or expression. Each action is associated with the value of an **integral constant expression**—any combination of character and integer constants that evaluates to a constant integer value.

Using a `switch` Statement to Count A, B, C, D and F Grades

Figure 4.6 calculates the class average of a set of numeric grades entered by the user. The `switch` statement determines each grade's letter equivalent (A, B, C, D or F) and increments the appropriate grade counter. The program also displays a summary of the number of students who received each grade.

```
1  // fig04_06.cpp
2  // Using a switch statement to count letter grades.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  int main() {
8     int total{0}; // sum of grades
9     int gradeCounter{0}; // number of grades entered
10    int aCount{0}; // count of A grades
11    int bCount{0}; // count of B grades
12    int cCount{0}; // count of C grades
13    int dCount{0}; // count of D grades
14    int fCount{0}; // count of F grades
15
16    cout << "Enter the integer grades in the range 0-100.\n"
```

```
17          << "Type the end-of-file indicator to terminate input:\n"
18          << " On UNIX/Linux/macOS type <Ctrl> d then press Enter\n"
19          << " On Windows type <Ctrl> z then press Enter\n";
20
21       int grade;
22
23       // loop until user enters the end-of-file indicator
24       while (cin >> grade) {
25          total += grade; // add grade to total
26          ++gradeCounter; // increment number of grades
27
28          // increment appropriate letter-grade counter
29          switch (grade / 10) {
30             case 9: // grade was between 90
31             case 10: // and 100, inclusive
32                ++aCount;
33                break; // exits switch
34
35             case 8: // grade was between 80 and 89
36                ++bCount;
37                break; // exits switch
38
39             case 7: // grade was between 70 and 79
40                ++cCount;
41                break; // exits switch
42
43             case 6: // grade was between 60 and 69
44                ++dCount;
45                break; // exits switch
46
47             default: // grade was less than 60
48                ++fCount;
49                break; // optional; exits switch anyway
50          } // end switch
51       } // end while
52
53       // set floating-point number format
54       cout << fixed << setprecision(2);
55
56       // display grade report
57       cout << "\nGrade Report:\n";
58
59       // if user entered at least one grade...
60       if (gradeCounter != 0) {
61          // calculate average of all grades entered
62          double average{static_cast<double>(total) / gradeCounter};
63
64          // output summary of results
65          cout << "Total of the " << gradeCounter << " grades entered is "
66             << total << "\nClass average is " << average
67             << "\nNumber of students who received each grade:"
68             << "\nA: " << aCount << "\nB: " << bCount << "\nC: " << cCount
69             << "\nD: " << dCount << "\nF: " << fCount << "\n";
70       }
71       else { // no grades were entered, so output appropriate message
```

```
72          cout << "No grades were entered" << "\n";
73       }
74    }
```

```
Enter the integer grades in the range 0-100.
Type the end-of-file indicator to terminate input:
   On UNIX/Linux/macOS type <Ctrl> d then press Enter
   On Windows type <Ctrl> z then press Enter
99
92
45
57
63
71
76
85
90
100
^Z

Grade Report:
Total of the 10 grades entered is 778
Class average is 77.80

Number of students who received each grade:
A: 4
B: 1
C: 2
D: 1
F: 2
```

Fig. 4.6 | Using a `switch` statement to count letter grades.

Figure 4.6 declares local variables `total` (line 8) and `gradeCounter` (line 9) to keep track of the sum of the grades entered by the user and the number of grades entered. Lines 10–14 declare and initialize to 0 counter variables for each grade category. Lines 24–51 input an arbitrary number of integer grades using sentinel-controlled iteration, update variables `total` and `gradeCounter`, and increment an appropriate letter-grade counter for each grade entered. Lines 54–73 output a report containing the total of all grades entered, the average grade and the number of students who received each letter grade.

Reading Grades from the User

Lines 16–19 prompt the user to enter integer grades or type the end-of-file indicator to terminate the input. The **end-of-file indicator** is a system-dependent keystroke combination used to indicate that there's no more data to input. In Chapter 8, you'll see how the end-of-file indicator is used when a program reads its input from a file.

The keystroke combinations for entering end-of-file are system dependent. On UNIX/Linux/macOS systems, type the sequence

```
<Ctrl> d
```

on a line by itself. This notation means to press both the *Ctrl* key and the *d* key simultaneously. On Windows systems, type

```
<Ctrl> z
```

On some systems, you must also press *Enter*. Also, Windows typically displays `^Z` on the screen when you type the end-of-file indicator, as shown in the output of Fig. 4.6.

The `while` statement (lines 24–51) obtains the user input. Line 24

```
while (cin >> grade) {
```

performs the input in the `while` statement's condition. In this case, the loop-continuation condition evaluates to `true` if `cin` successfully reads an `int` value. If the user enters the end-of-file indicator, the condition evaluates to `false`.

If the condition is `true`, line 25 adds `grade` to `total`, and line 26 increments `grade-Counter`. These are used to compute the average. Next, lines 29–50 use a `switch` statement to increment the appropriate letter-grade counter based on the numeric grade entered.

Processing the Grades

The `switch` statement (lines 29–50) determines which counter to increment. We assume that the user enters a valid grade in the range 0–100. A grade in the range 90–100 represents A, 80–89 represents B, 70–79 represents C, 60–69 represents D and 0–59 represents F. The `switch` statement's block contains a sequence of **`case` labels** and an optional **`default` case**, which can appear anywhere in the `switch`, but normally appears last. These are used in this example to determine which counter to increment based on the grade.

When the flow of control reaches the `switch`, the program evaluates the **controlling expression** in the parentheses (`grade / 10`) following keyword `switch`. The program compares this expression's value with each `case` label. The expression must have a signed or unsigned integral type—`bool`, `char`, `char8_t`, `char16_t`, `char32_t`, `wchar_t`, `int`, `long` or `long long`.

The controlling expression in line 29 performs integer division, which truncates the fractional part of the result. When we divide a value from 0 to 100 by 10, the result is always a value from 0 to 10. We use several of these values in our `case` labels. If the user enters the integer `85`, the controlling expression evaluates to 8. The `switch` compares `8` with each `case` label. If a match occurs (`case 8:` at line 35), that `case`'s statements execute. For `8`, line 36 increments `bCount`, because a grade in the 80s is a B. The **`break` statement** (line 37) exits the `switch`. In this program, we reach the end of the `while` loop, so control returns to the loop-continuation condition in line 24 to determine whether the loop should continue executing.

The `case`s in our `switch` explicitly test for the values `10`, `9`, `8`, `7` and `6`. Note the cases at lines 30–31 that test for the values `9` and `10` (both of which represent the grade A). Listing cases consecutively in this manner with no statements between them enables the cases to perform the same set of statements—when the controlling expression evaluates to `9` or `10`, the statements in lines 32–33 execute.

The `switch` statement does not provide a mechanism for testing ranges of values, so every value you need to test must be listed in a separate `case` label. Each `case` can have multiple statements.

The `switch` statement differs from other control statements in that it does not require braces around multiple statements in a `case`, unless you need to declare a variable in a `case`.

`case` without a `break` Statement—C++17 `[[fallthrough]]` Attribute

Without `break` statements, each time a match occurs in the `switch`, the statements for that `case` and subsequent `case`s execute until a `break` statement or the end of the `switch` is reached. This is referred to as "falling through" to the statements in subsequent `case`s.[4]

4. This feature is perfect for writing a concise program that displays the iterative song “The Twelve Days of Christmas.” As an exercise, you might write the program, then use one of the many free, open-source text-to-speech programs to speak the song. You might also tie your program to a free, open-source MIDI (“Musical Instrument Digital Interface”) program to create a singing version of your program accompanied by music.

Err 17 Forgetting a `break` statement when one is needed is a logic error. To call your attention to this possible problem, many compilers issue a warning when a `case` label is followed by one or more statements and does not contain a `break` statement. For such instances in which “falling through” is the desired behavior, C++17 introduced the `[[fallthrough]]` attribute. You can tell the compiler that “falling through” to the next `case` is the correct behavior by placing the statement

```
[[fallthrough]];
```

where the `break` statement would normally appear.

The `default` Case

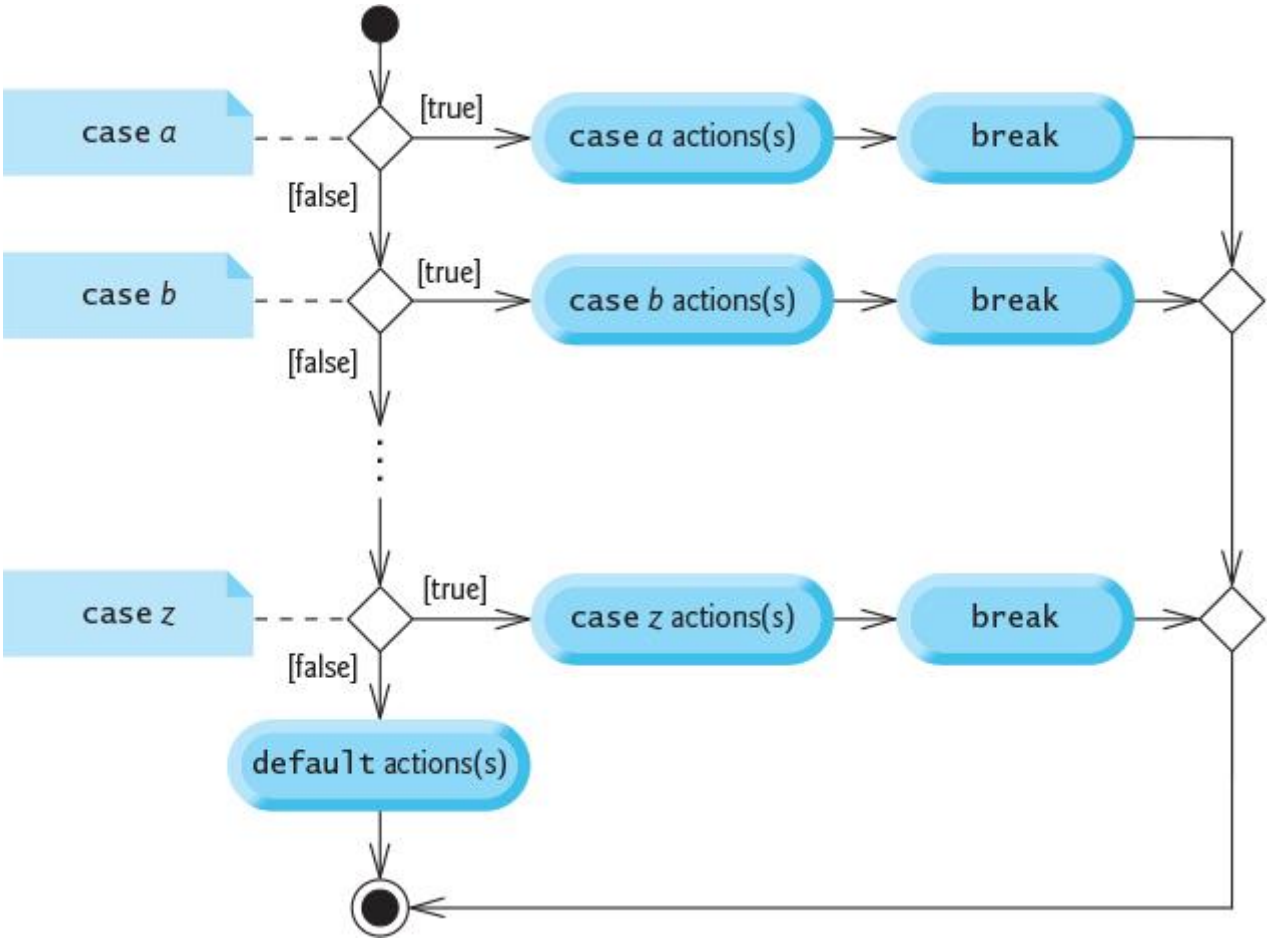
If no match occurs between the controlling expression’s value and any of the `case` labels, the `default` case (lines 47–49) executes. We use the `default` case in this example to process all controlling-expression values that are less than `6`—that is, all failing grades. If no match occurs and the `switch` does not contain a `default` case, program control simply continues with the first statement after the `switch`. In a `switch`, it’s good practice to test for all possible values of the controlling expression.

Displaying the Grade Report

Lines 54–73 output a report based on the grades entered. Line 60 determines whether the user entered at least one grade—this helps us avoid dividing by zero, which for integer division causes the program to fail and for floating-point division produces the value `nan`—for “not a number.” If so, line 62 calculates the average of the grades. Lines 65–69 then output the total of all the grades, the class average and the number of students who received each letter grade. If no grades were entered, line 72 outputs an appropriate message. The output in Fig. 4.6 shows a sample grade report based on 10 grades.

`switch` Statement UML Activity Diagram

The following is the UML activity diagram for the general `switch` statement:



Most `switch` statements use a `break` in each `case` to terminate the `switch` after the `case` is processed. The diagram emphasizes this by including `break` statements and showing that the `break` at the end of a `case` causes control to exit the `switch` statement immediately.

The `break` statement is not required for the `switch`'s last case (or the optional `default` case, when it appears last), because execution continues with the next statement after the `switch`. Provide a `default` case in every `switch` statement to focus you on processing exceptional conditions.

Notes on `case`s

Each `case` in a `switch` statement must contain a constant integral expression—that is, any expression that evaluates to a constant integer value. You also can use `enum` constants (introduced in Section 5.9) and **character literals**—specific characters in single quotes, such as `'A'`, `'7'` or `'$'`, which represent the integer values of characters. (Appendix B shows the integer values of the characters in the ASCII character set, which is a subset of the Unicode character set.)

In Chapter 10, OOP: Inheritance and Runtime Polymorphism, we present a more elegant way to implement `switch` logic. We use a technique called polymorphism to create programs that are often clearer, easier to maintain and easier to extend than programs using `switch` logic.

17 4.9 C++17 Selection Statements with Initializers

17 Earlier, we introduced the `for` iteration statement. In the `for` header's initialization section, we declared and initialized a control variable, which limited that variable's scope to the `for` statement. C++17's **selection statements with initializers** enable you to include variable initializers before the condition in an `if` or `if`...`else` statement and before the controlling expression of a `switch` statement. As with the `for` statement, these variables are known only in the statements where they're declared. Figure 4.7 shows `if`...`else` statements with initializers. We'll use

both `if`...`else` and `switch` statements with initializers in Fig. 5.5, which implements a popular casino dice game.

```
1  // fig04_07.cpp
2  // C++17 if statements with initializers.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     if (int value{7}; value == 7) {
8        cout << "value is " << value << "\n";
9     }
10    else {
11       cout << "value is not 7; it is " << value << "\n";
12    }
13
14    if (int value{13}; value == 9) {
15       cout << "value is " << value << "\n";
16    }
17    else {
18       cout << "value is not 9; it is " << value << "\n";
19    }
20 }
```

```
value is 7
value is not 9; it is 13
```

Fig. 4.7 C++17 `if` statements with initializers.

Syntax of Selection Statements with Initializers

For an `if` or `if`...`else` statement, you place the initializer first in the condition's parentheses. For a `switch` statement, you place the initializer first in the controlling expression's parentheses. The initializer must end with a semicolon (`;`), as in lines 7 and 14. The initializer can declare multiple variables of the same type in a comma-separated list.

Scope of Variables Declared in the Initializer

Any variable declared in the initializer of an `if`, `if`...`else` or `switch` statement may be used throughout the remainder of the statement. In lines 7–12, we use the variable `value` to determine which branch of the `if`...`else` statement to execute, then use `value` in the output statements of both branches. When the `if`...`else` statement terminates, `value` no longer exists, so we can use that identifier again in the second `if`...`else` statement to declare a new variable known only in that statement.

To prove that `value` is not accessible outside the `if`...`else` statements, we provided a second version of this program (`fig04_07_with_error.cpp`) that attempts to access variable `value` after (and thus outside the scope of) the second `if`...`else` statement. This produces the following compilation errors in our three compilers:

- Visual Studio: `'value': undeclared identifier`
- Xcode: `error: use of undeclared identifier 'value'`
- GNU g++: `error: 'value' was not declared in this scope`

4.10 `break` and `continue` Statements

In addition to selection and iteration statements, C++ provides `break` and `continue` statements to alter the flow of control. The preceding section showed how `break` could be used to terminate a `switch` statement's execution. This section discusses how to use `break` in iteration statements.

`break` Statement

Executing a `break` statement in a `while`, `for`, `do...while` or `switch` causes immediate exit from that statement—execution continues with the first statement after the control statement. Common uses of `break` include escaping early from a loop or exiting a `switch` (as in Fig. 4.6). Figure 4.8 demonstrates a `break` statement exiting early from a `for` statement.

```
1  // fig04_08.cpp
2  // break statement exiting a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     int count; // control variable also used after loop
8
9     for (count = 1; count <= 10; ++count) { // loop 10 times
10       if (count == 5) {
11          break; // terminates for loop if count is 5
12       }
13
14       cout << count << " ";
15    }
16
17    cout << "\nBroke out of loop at count = " << count << "\n";
18 }
```

```
1 2 3 4
Broke out of loop at count = 5
```

Fig. 4.8 `break` statement exiting a `for` statement.

When the `if` statement nested at lines 10–12 in the `for` statement (lines 9–15) detects that `count` is `5`, the `break` statement at line 11 executes. This terminates the `for` statement, and the program proceeds to line 17 (immediately after the `for` statement), which displays a message indicating the value of the control variable when the loop terminated. The loop fully executes its body only four times instead of 10. Note that we could have initialized `count` in line 7 and left the `for` header's initialization section empty, as in:

```
for (; count <= 10; ++count) { // loop 10 times
```

`continue` Statement

Executing the `continue` statement in a `while`, `for` or `do...while` skips the remaining statements in the loop body and proceeds with the next iteration of the loop. In `while` and `do...while` statements, the program evaluates the loop-continuation test immediately after the `continue` statement executes. In a `for` statement, the increment expression executes, then the program evaluates the loop-continuation test.

```
1  // fig04_09.cpp
2  // continue statement terminating an iteration of a for statement.
3  #include <iostream>
4  using namespace std;
5
6  int main() {
7     for (int count{1}; count <= 10; ++count) { // loop 10 times
8        if (count == 5) {
9           continue; // skip remaining code in loop body if count is 5
10       }
11
12       cout << count << " ";
13    }
14
15    cout << "\nUsed continue to skip printing 5" << "\n";
16 }
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing 5
```

Fig. 4.9 `continue` statement terminating an iteration of a `for` statement.

Figure 4.9 uses `continue` (line 9) to skip the statement at line 12 when the nested `if` determines that `count`'s value is `5`. When the `continue` statement executes, program control continues with the increment of the control variable in the `for` statement (line 7).

Some programmers feel that `break` and `continue` violate structured programming. Since the same effects are achievable with structured-programming techniques, these programmers prefer to avoid `break` or `continue`.

Perf There's a tension between achieving quality software engineering and achieving the best-performing software. Sometimes one of these goals is achieved at the expense of the other. For all but the most performance-intensive situations, you should first make your code simple and correct, then make it fast and small—but only if necessary.

4.11 Logical Operators

The conditions in `if`, `if`...`else`, `while`, `do`...`while` and `for` statements determine how to continue a program's flow of control. So far, we've studied only simple conditions, such as `count <= 10`, `number != sentinelValue` and `total > 1000`. Simple conditions are expressed with the relational operators `>`, `<`, `>=` and `<=` and the equality operators `==` and `!=`. Each tests one condition. Sometimes control statements require more complex conditions to determine a program's flow of control. C++'s **logical operators** enable you to combine simple conditions. The logical operators are `&&` (logical AND), `||` (logical OR) and `!` (logical negation).

4.11.1 Logical AND (`&&`) Operator

Suppose we wish to ensure at some point in a program that two conditions are both true before we choose a certain path of execution. In this case, we can use the `&&` (**logical AND**) operator, as follows:

```
if (gender == FEMALE && age >= 65) {
   ++seniorFemales;
}
```

Assume `FEMALE` is a constant variable. This `if` statement contains two simple conditions. The condition `gender == FEMALE` determines whether a person is female. The condition `age >= 65` might be

evaluated to determine whether a person is a senior citizen. The `if` statement considers the combined condition

```
gender == FEMALE && age >= 65
```

which is true if and only if both simple conditions are true. In this case, the `if` statement's body increments `seniorFemales` by `1`. If either or both of the simple conditions are false, the program skips the increment. Some programmers find that the preceding combined condition is more readable when redundant parentheses are added, as in

```
(gender == FEMALE) && (age >= 65)
```

The following **truth table** summarizes the `&&` operator, showing all four possible combinations of the `bool` values `false` and `true` for *expression1* and *expression2*. C++ evaluates to zero (false) or nonzero (true) all expressions that include relational operators, equality operators or logical operators:

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

4.11.2 Logical OR (||) Operator

Now suppose we wish to ensure that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the `||` (**logical OR**) operator, as in the following program segment:

```
if ((semesterAverage >= 90) || (finalExam >= 90)) {
   cout << "Student grade is A\n";
}
```

This statement also contains two simple conditions. The condition `semesterAverage >= 90` determines whether the student deserves an A in the course for a solid performance throughout the semester. The condition `finalExam >= 90` determines whether the student deserves an A in the course for an outstanding performance on the final exam. The `if` statement then considers the combined condition

```
(semesterAverage >= 90) || (finalExam >= 90)
```

and awards the student an A if either or both of the simple conditions are true. The only time the message `"Student grade is A"` is not printed is when both of the simple conditions are false. The following is the truth table for the operator logical OR (`||`):

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Operator `&&` has higher precedence than operator `||`.[5] Both operators group left-to-right.

---

5. In general, use parentheses if there is ambiguity about evaluation order.

4.11.3 Short-Circuit Evaluation

The parts of an expression containing `&&` or `||` operators are evaluated only until it's known whether the condition is true or false. Thus, evaluation of the expression

```
(gender == FEMALE) && (age >= 65)
```

stops immediately if `gender` is not equal to `FEMALE` (i.e., the entire expression is false) and continues if `gender` is equal to `FEMALE` (i.e., the entire expression could still be true if the condition `age >= 65` is true). This feature of logical AND and logical OR expressions is called **short-circuit evaluation**.

In expressions using operator `&&`, a condition—we'll call this the dependent condition—may require another condition to be true for the evaluation of the dependent condition to be meaningful. In this case, the dependent condition should be placed after the `&&` operator to prevent errors. Consider the expression `(i != 0) && (10 / i == 2)`. The dependent condition `(10 / i == 2)` must appear after the `&&` operator to prevent the possibility of division by zero.

4.11.4 Logical Negation (!) Operator

The `!` (**logical negation**, also called **logical NOT** or **logical complement**) operator "reverses" the meaning of a condition. Unlike the logical operators `&&` and `||`, which are binary operators that combine two conditions, the logical negation operator is a unary operator that has only one condition as an operand. To execute code only when a condition is false, place the logical negation operator *before* the original condition, as in the program segment

```
if (!(grade == sentinelValue)) {
   cout << "The next grade is " << grade << "\n";
}
```

which executes the body statement only if `grade` is *not* equal to `sentinelValue`. The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has higher precedence than the equality operator.

In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational or equality operator. For example, the previous statement may also be written in a more readable manner as

```
if (grade != sentinelValue) {
   cout << "The next grade is " << grade << "\n";
}
```

This flexibility can help you express a condition more conveniently. The following is the truth table for the logical negation operator:

expression	!expression
false	true
true	false

4.11.5 Example: Producing Logical-Operator Truth Tables

Figure 4.10 uses logical operators to produce the truth tables discussed in this section. The output shows each expression that's evaluated and its `bool` result. By default, `bool` values `true` and `false` are displayed by `cout` and the stream-insertion operator as `1` and `0`, respectively, but the format function displays the word "true" or the word "false." Lines 10–14, 17–21 and 24–26 produce the truth tables for `&&`, `||` and `!`, respectively.

```
1  // fig04_10.cpp
2  // Logical operators.
3  #include <iostream>
4  #include <fmt/format.h> // in C++20, this will be #include <format>
5  using namespace std;
6  using namespace fmt; // not needed in C++20
7
8  int main() {
9     // create truth table for && (logical AND) operator
10    cout << "Logical AND (&&)\n"
11       << format("false && false: {}\n", false && false)
12       << format("false && true: {}\n", false && true)
13       << format("true && false: {}\n", true && false)
14       << format("true && true: {}\n\n", true && true);
15
16    // create truth table for || (logical OR) operator
17    cout << "Logical OR (||)\n"
18       << format("false || false: {}\n", false || false)
```

```
19          << format("false || true: {}\n", false || true)
20          << format("true || false: {}\n", true || false)
21          << format("true || true: {}\n\n", true || true);
22
23       // create truth table for ! (logical negation) operator
24       cout << "Logical negation (!)\n"
25          << format("!false: {}\n", !false)
26          << format("!true: {}\n", !true);
27    }
```

```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical negation (!)
!false: true
!true: false
```

Fig. 4.10 Logical operators.

Precedence and Grouping of the Operators Presented So Far

The following table shows the precedence and grouping of the C++ operators introduced so far—from top to bottom in decreasing order of precedence:

Operators	Grouping
`++ -- static_cast<type>()`	left to right
`++ -- + - !`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&&`	left to right

Operators	Grouping
`||`	left to right
`?:`	right to left
`=  +=  -=  *=  /=  %=`	right to left
`,`	left to right

Err 4.12 Confusing the Equality (==) and Assignment (=) Operators

There's one logic error that C++ programmers, no matter how experienced, tend to make so frequently that we feel it requires a separate section. That error is accidentally swapping the operators `==` (equality) and `=` (assignment). What makes this so damaging is that it ordinarily does not cause compilation errors. Statements with these errors tend to compile correctly and run to completion, often generating incorrect results through runtime logic errors. Today's compilers generally can issue warnings when `=` is used in contexts where `==` is expected (see the end of this section for details on enabling this).

Two aspects of C++ contribute to these problems. One is that any expression that produces a value can be used in the decision portion of any control statement. If the expression's value is zero, it's treated as `false`. If the value is nonzero, it's treated as `true`. The second is that assignments produce a value—namely, the value of the variable on the assignment operator's left side. For example, suppose we intend to write

```
if (payCode == 4) { // good
   cout << "You get a bonus!" << "\n";
}
```

but we accidentally write

```
if (payCode = 4) { // bad
   cout << "You get a bonus!" << "\n";
}
```

The first `if` statement properly awards a bonus to the person whose `payCode` is equal to 4. The second one—which contains the error—evaluates the assignment expression in the `if` condition to the constant 4. Any nonzero value is `true`, so this condition always evaluates as `true` and the person always receives a bonus regardless of the pay code! Even worse, the pay code has been modified when it was only supposed to be examined!

lvalues and rvalues

You can prevent this problem with a simple trick. First, it's helpful to know what's allowed to the left of an assignment operator. Variable names are said to be **lvalues** (for "left values") because they can

be used on an assignment operator's left side. Literals are said to be **rvalues** (for "right values")—they can be used on only an assignment operator's right side. You also can use *lvalues* as *rvalues* on an assignment's right side, but not vice versa.

Programmers normally write conditions such as `x == 7` with the variable name (an *lvalue*) on the left and the literal (an *rvalue*) on the right. Placing the literal on the left, as in `7== x` (which is syntactically correct and is sometimes called a "Yoda condition"[6]), enables the compiler to issue an error if you accidentally replace the `==` operator with `=`. The compiler treats this as a compilation error because you can't change a literal's value.

6. "Yoda conditions." Accessed November 19, 2021. https://en.wikipedia.org/wiki/Yoda_conditions.

Using `==` in Place of `=`

There's another equally unpleasant situation. Suppose you want to assign a value to a variable with a simple statement like

```
x = 1;
```

but instead write

```
x == 1;
```

Here, too, this is not a syntax error. Rather, the compiler simply evaluates the expression. If `x` is equal to `1`, the condition is true, and the expression evaluates to a nonzero (true) value. If `x` is not equal to `1`, the condition is false and the expression evaluates to 0. Regardless of the expression's value, there's no assignment operator, so the value is lost. The value of `x` remains unaltered, probably causing an execution-time logic error. Using operator `==` for assignment and using operator `=` for equality are logic errors. Use your text editor to search for all occurrences of `=` in your program and check that you have the correct assignment, relational or equality operator in each place.

Enabling Warnings

Xcode automatically issues a warning when you use `=` where `==` is expected. Some compilers require you to enable warnings before they'll issue warning messages. For GNU `g++`, add the `-Wall` (enable all warnings) flag to your compilation command—see the `g++` documentation for details on enabling subsets of the potential warnings. For Visual C++:

1. In your solution, right-click the project's name and select **Properties**.
2. Expand **Code Analysis** and select **General**.
3. For **Enable Code Analysis on Build**, select **Yes**, then click **OK**.

4.13 Objects-Natural Case Study: Using the `miniz-cpp` Library to Write and Read ZIP files[7]

7. This example does not compile in GNU C++.

Perf **Data compression** reduces the size of data—typically to save memory, to save secondary storage space or to transmit data over the Internet faster by reducing the number of bytes. **Lossless data-compression algorithms** compress data in a manner that does not lose information—the data

can be uncompressed and restored to its original form. **Lossy data-compression algorithms** permanently discard information. Such algorithms are often used to compress images, audio and video. For example, when you watch streaming video online, the video is often compressed ahead of time using a lossy algorithm to minimize the total bytes transferred over the Internet. Though some of the video data is discarded, a lossy algorithm compresses the data in a manner such that most people do not notice the removed information as they watch the video. The video quality is still "pretty good."

ZIP Files

You've probably used ZIP files—if not, you almost certainly will. The **ZIP file format**[8] is a lossless compression[9] format that has been in use for over 30 years. Lossless compression algorithms use various techniques for compressing data—such as

8. "Zip (file format)." Accessed November 19, 2021. https://en.wikipedia.org/wiki/Zip_(file_format).
9. "Data compression." Accessed November 19, 2021. https://en.wikipedia.org/wiki/Data_compression#Lossless.

- replacing duplicate patterns, such as text strings in a document or pixels in an image, with references to a single copy, and
- replacing a group of image pixels that have the same color with one pixel of that color and a count (known as "run-length encoding").

ZIP is used to compress files and directories into a single file, known as an **archive file**. ZIP files are often used to distribute software faster over the Internet. Today's operating systems typically have built-in support for creating ZIP files and extracting their contents.

Open-Source `miniz-cpp` Library

Many open-source libraries support programmatic manipulation of ZIP archive files and other popular archive-file formats, such as TAR, RAR and 7-Zip.[10] Figure 4.11 continues our Objects-Natural presentation by using objects of the open-source `miniz-cpp`[11,12] library's class `zip_file` to create and read ZIP files. The `miniz-cpp` library is a "header-only library"—it's defined in header file `zip_file.hpp`, which you can simply place in the same folder as this example and include the header in your program (line 5). We provide the library in the `examples` folder's `libraries/miniz-cpp` subfolder. Header files are discussed in depth in Chapter 9.

10. "List of archive formats." Wikipedia. Wikimedia Foundation, March 19, 2020. https://en.wikipedia.org/wiki/List_of_archive_formats.
11. https://github.com/tfussell/miniz-cpp.
12. The `miniz-cpp` library provides capabilities nearly identical to the Python standard library's `zipfile` module (https://docs.python.org/3/library/zipfile.html), so the `miniz-cpp` GitHub repository refers you to that documentation page for the list of features.

```
1  // fig04_11.cpp
2  // Using the miniz-cpp header-only library to write and read a ZIP file.
3  #include <iostream>
4  #include <string>
5  #include "zip_file.hpp"
6  using namespace std;
7
```

Fig. 4.11 Using the `miniz-cpp` header-only library to write and read a ZIP file.

Inputting a Line of Text from the User with `getline`

The `getline` function call reads all the characters you type until you press *Enter*:

```
 8   int main() {
 9      cout << "Enter a ZIP file name: ";
10      string zipFileName;
11      getline(cin, zipFileName); // inputs a line of text
12
```

```
Enter a ZIP file name: c:\users\useraccount\Documents\test.zip
```

Here we use `getline` to read from the user the location and name of a file, and store it in the `string` variable `zipFileName`. Like class `string`, `getline` requires the `<string>` header and belongs to namespace `std`.

Creating Sample Content to Write an Individual File in the ZIP File

The following statement creates a lengthy string named `content` consisting of sentences from this chapter's introduction:

```
13   // string literals separated only by whitespace are combined
14   // into a single string by the compiler
15   string content{
16       "This chapter introduces all but one of the remaining control "
17       "statements--the for, do...while, switch, break and continue "
18       "statements. We explore the essentials of counter-controlled "
19       "iteration. We use compound-interest calculations to begin "
20       "investigating the issues of processing monetary amounts. First, "
21       "we discuss the representational errors associated with "
22       "floating-point types. We use a switch statement to count the "
23       "number of A, B, C, D and F grade equivalents in a set of "
24       "numeric grades. We show C++17's enhancements that allow you to "
25       "initialize one or more variables of the same type in the "
26      "headers of if and switch statements."};
27
```

We'll use the `miniz-cpp` library to write this string as a text file that will be compressed into a ZIP file. Each string literal in the preceding statement is separated from the next only by whitespace. The C++ compiler automatically assembles such string literals into a single string literal, which we use to initialize the `string` variable `content`. The following statement outputs the length of `content` (632 bytes).

```
28      cout << "\ncontent.length(): " << content.length();
29
```

```
content.length(): 632
```

Creating a `zip_file` Object

The `miniz-cpp` library's `zip_file` class—located in the library's `miniz_cpp` namespace—is used to create a ZIP file. The statement

```
30      miniz_cpp::zip_file output; // create zip_file object
31
```

creates the `zip_file` object `output`, which will perform the ZIP operations to create the archive file.

Creating a File in the `zip_file` Object and Saving That Object to Disk

Line 33 calls `output`'s `writestr` member function, which creates one file (`"intro.txt"`) in the ZIP archive containing the text in `content`. Line 34 calls `output`'s `save` member function to store the `output` object's contents in the file specified by `zipFileName`:

```
32      // write content into a text file in output
33      output.writestr("intro.txt", content); // create file in ZIP
34      output.save(zipFileName); // save output to zipFileName
35
```

ZIP Files Appear to Contain Random Symbols

ZIP is a binary format, so if you open the compressed file in a text editor, you'll see mostly gibberish. Below is what the file looks like in the Windows Notepad text editor:

```
PK□□   □ □†—P□VÉ´o□  y         intro.txt]'KnÜ0♠†÷>□÷±
tõu›"'(Ð5-Ó#¢²ä'ÒL'ÇÈ‰CÊƒ ÈÒ&éï□øwd…□ñ¨$À¹JY[ □L  -V¡d,²A□□B;ræ|P|1
Z±ÒN¹ê4ùÆVd„µÌó|‹œh□½q
q„E□ÿ æµŸrnôév†?□ôÿHE¨ƒHÕÞ3&ut(-› é□M´□lÏX¹ä~Ú”li?loØwI+□L¡¥¾¥P□,tá<p¾Ú/öÞ|8ŒU□uÐ!Åœ«□v³]Qî
€»ÓMâ□‹Öq.□¬¬¡©>"9♠fb;□□H□KO□6ƒ+X qØRéÄé(&□êý ýÐŽ˜à#□×Ûm;dÈm_¬Søc„Ÿ#<ð«gù □ÁÕòú×øŠÉ³´
ý{dÅm~HÂáÜ:y□Ë
žŸž¾}□S 1‡³□#aõÖm~/Í□XVì%ð+□X,ÖÐ□…qIgf,âNÝ'Ã]p$ãIŸóÖ•~5¨óð□PK□   □   □ □†—P□VÉ´o□  y
intro.txtPK□    □ □ 7   –□
```

Reading the Contents of the ZIP File

You can locate the ZIP file on your system and extract (decompress) its contents to confirm that the ZIP file was written correctly. The `miniz-cpp` library also supports reading and processing a ZIP file's contents programmatically. The following statement creates a `zip_file` object named `input` and initializes it with the name of a ZIP file:

```
36      miniz_cpp::zip_file input{zipFileName}; // load zipFileName
37
```

This reads the corresponding ZIP archive's contents. We can then use the `zip_file` object's member functions to interact with the archived files.

Displaying the Name and Contents of the ZIP File

The following statements call `input`'s `get_filename` and `printdir` member functions to display the ZIP's file name and a directory listing of the ZIP file's contents, respectively.

```
38      // display input's file name and directory listing
39      cout << "\n\nZIP file's name: " << input.get_filename()
40         << "\n\nZIP file's directory listing:\n";
41      input.printdir();
42
```

```
ZIP file's name: c:\users\useraccount\Documents\test.zip

ZIP file's directory listing:
  Length      Date    Time    Name
---------  ---------- -----   ----
      632  11/28/2021 16:48   intro.txt
---------                     -------
      632                     1 file
```

The output shows that the ZIP archive contains the file `intro.txt` and that the file's length is `632`, which matches that of the string `content` we wrote to the file earlier.

Getting and Displaying Information About a Specific File in the ZIP Archive

Line 44 declares and initializes the `zip_info` object `info`:

```
43      // display info about the compressed intro.txt file
44      miniz_cpp::zip_info info{input.getinfo("intro.txt")};
45
```

Calling `input`'s `getinfo` member function returns a `zip_info` object (from namespace `miniz_cpp`) for the specified file in the archive. Sometimes objects expose data so that you can access it directly using the object's name and a dot (`.`) operator. For example, the object `info` contains information about the archive's `intro.txt` file, including the file's name (`info.filename`), its uncompressed size (`info.file_size`) and its compressed size (`info.compress_size`):

```
46      cout << "\nFile name: " << info.filename
47         << "\nOriginal size: " << info.file_size
48         << "\nCompressed size: " << info.compress_size;
49
```

```
File name: intro.txt
Original size: 632
Compressed size: 360
```

Note that `intro.txt`'s compressed size is 360 bytes—43% smaller than the original file. Compression amounts vary considerably, based on the type of content being compressed.

Extracting `"intro.txt"` and Displaying Its Original Contents

You can extract the original contents of a compressed file from the ZIP archive. Here we use the `input` object's `read` member function, passing the `zip_info` object (`info`) as an argument. This returns as a `string` the contents of the file represented by the object `info`:

```
50   // original file contents
51   string extractedContent{input.read(info)};
52
```

We output `extractedContent` to show that it matches the original string `content` that we "zipped up." This was indeed a lossless compression:

```
53      cout << "\n\nOriginal contents of intro.txt:\n"
54         << extractedContent << "\n";
55   }
```

```
Original contents of intro.txt:
This chapter introduces all but one of the remaining control statements--the
for, do...while, switch, break and continue statements. We explore the
essentials of counter-controlled iteration. We use compound-interest
calculations to begin investigating the issues of processing monetary
amounts. First, we discuss the representational errors associated with
floating-point types. We use a switch statement to count the number of A, B,
C, D and F grade equivalents in a set of numeric grades. We show C++17's
enhancements that allow you to initialize one or more variables of the same
type in the headers of if and switch statements.
```

20 4.14 C++20 Text Formatting with Field Widths and Precisions

Section 3.13 introduced C++20's `format` function (in header `<format>`), which provides powerful new text-formatting capabilities. Figure 4.12 shows how `format` strings can concisely specify what each value's format should be. We reimplement the formatting introduced in Fig. 4.4's compound-interest problem. Figure 4.12 produces the same output as Fig. 4.4, so we'll focus exclusively on the format strings in lines 13, 14, 17 and 22.

```
1   // fig04_12.cpp
2   // Compound-interest example with C++20 text formatting.
3   #include <iostream>
4   #include <cmath> // for pow function
5   #include <fmt/format.h> // in C++20, this will be #include <format>
6   using namespace std;
7   using namespace fmt; // not needed in C++20
8
9   int main() {
10     double principal{1000.00}; // initial amount before interest
11     double rate{0.05}; // interest rate
12
13     cout << format("Initial principal: {:>7.2f}\n", principal)
14        << format(" Interest rate: {:>7.2f}\n", rate);
15
16     // display headers
17     cout << format("\n{}{:>20}\n", "Year", "Amount on deposit");
18
19     // calculate amount on deposit for each of ten years
20     for (int year{1}; year <= 10; ++year) {
21        double amount = principal * pow(1.0 + rate, year);
22        cout << format("{:>4d}{:>20.2f}\n", year, amount);
23     }
24  }
```

```
Initial principal: 1000.00
   Interest rate:     0.05
```

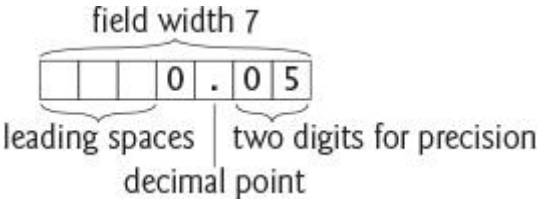
```
Year   Amount on deposit
   1             1050.00
   2             1102.50
   3             1157.63
   4             1215.51
   5             1276.28
   6             1340.10
   7             1407.10
   8             1477.46
   9             1551.33
  10             1628.89
```

Fig. 4.12 Compound-interest example with C++20 string formatting.

Formatting the Principal and Interest Rate

The `format` calls in lines 13 and 14 each use the placeholder `{:>7.2f}` to format the values of `principal` and `rate`. A colon (`:`) in a placeholder introduces a **format specifier** that indicates how a corresponding value should be formatted. The format specifier `>7.2f` is for a floating-point number (`f`) that should be **right-aligned (>)** in a `7`-character field width that has two digits of precision (`.2`) to the right of the decimal point. Unlike `setprecision` and `fixed` shown earlier, format settings specified in placeholders are not "sticky"—they apply only to the value that's inserted into that placeholder.

The value of `principal` (`1000.00`) requires exactly seven characters to display, so no spaces are required to fill out the field width. The value of `rate` (`0.05`) requires only four total character positions, so it will be right-aligned in the field of seven characters and filled from the left with leading spaces, as in



Numeric values are right-aligned by default, so the > is not required here. You can **left-align** numeric values in a field width via <.

Formatting the Year and Amount-on-Deposit Column Heads

In line 17's format string

```
"\n{}{:>20}\n"
```

the string `"Year"` is simply placed at the position of the first placeholder, which does not contain a format specifier. The second placeholder indicates that `"Amount on Deposit"` (17 characters) should be right-aligned (`>`) in a field of `20` characters—`format` inserts three leading spaces to right-align the string. Strings are left-aligned by default, so the > is required here to force right-alignment.

Formatting the Year and Amount-on-Deposit Values in the `for` Loop

The format string in line 22

```
"{:>4d}{:>20.2f}\n"
```

uses two placeholders to format the loop's output. The placeholder `{:>4d}` indicates that `year`'s value should be formatted as an integer (`d` means decimal integer) right-aligned (`>`) in a field of width `4`. This right-aligns all the year values under the `"Year"` column.

The placeholder `{:>20.2f}` formats `amount`'s value as a floating-point number (`f`) right-aligned (`>`) in a field width of `20` with a decimal point and two digits to the right of the decimal point (`.2`). Formatting the `amount`s this way *aligns their decimal points vertically*, as is typical with monetary amounts. The field width of 20 right-aligns the `amount`s under `"Amount on Deposit"`.

4.15 Wrap-Up

In this chapter, we completed our introduction to all but one of C++'s control statements, which enable you to control the flow of execution in functions. Chapter 3 discussed `if`, `if`...`else` and `while`. Chapter 4 demonstrated `for`, `do`...`while` and `switch`. We showed C++17's enhancements that allow you to initialize a variable in the header of an `if` and `switch` statement. You used the `break` statement to exit a `switch` statement and to terminate a loop immediately. You used a `continue` statement to terminate a loop's current iteration and proceed with the loop's next iteration. We introduced C++'s logical operators, which enable you to use more complex conditional expressions in control statements.

In the Objects-Natural case study, we used the `miniz-cpp` open-source library to create and read compressed ZIP archive files. Finally, we introduced more of C++20's powerful and expressive text-formatting features. In Chapter 5, you'll create your own custom functions.

5. Functions and an Intro to Function Templates

Objectives

In this chapter, you'll:

- Construct programs modularly from functions.
- Use common math library functions and learn about math functions and constants added in C++20, C++17 and C++11.
- Declare functions with function prototypes.
- View many key C++ standard library headers.
- Use random numbers to implement game-playing apps.
- Declare constants in scoped `enum`s and use constants without their type names via C++20's `using enum` declarations.
- Understand the scope of identifiers.
- Use inline functions, references and default arguments.
- Define overloaded functions that handle a variety of different argument types.
- Define function templates that can generate families of overloaded functions.
- Write and use recursive functions.
- Zajnropc vrq lnfylun lhqtomh uyqmmhzg tupb j dvql psrpu iw dmwwqnddwjqz (see Section 5.20).

Outline

5.1 Introduction
5.2 C++ Program Components
5.3 Math Library Functions
5.4 Function Definitions and Function Prototypes
5.5 Order of Evaluation of a Function's Arguments
5.6 Function-Prototype and Argument-Coercion Notes
5.6.1 Function Signatures and Function Prototypes
5.6.2 Argument Coercion
5.6.3 Argument-Promotion Rules and Implicit Conversions
5.7 C++ Standard Library Headers
5.8 Case Study: Random-Number Generation
5.8.1 Rolling a Six-Sided Die
5.8.2 Rolling a Six-Sided Die 60,000,000 Times
5.8.3 Seeding the Random-Number Generator
5.8.4 Seeding the Random-Number Generator with `random_device`
5.9 Case Study: Game of Chance; Introducing Scoped `enum`s
5.10 Scope Rules
5.11 Inline Functions
5.12 References and Reference Parameters
5.13 Default Arguments
5.14 Unary Scope Resolution Operator
5.15 Function Overloading
5.16 Function Templates
5.17 Recursion
5.18 Example Using Recursion: Fibonacci Series

5.19 Recursion vs. Iteration
5.20 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz
5.21 Wrap-Up

5.1 Introduction

20 17 11 In this chapter, we introduce custom function definitions. We overview some C++ standard library math functions and introduce new functions and constants added in C++20, C++17 and C++11. We introduce function prototypes and discuss how the compiler uses them, if necessary, to convert the type of an argument in a function call to the type specified in a function's parameter list. We also present an overview of the C++ standard library's headers.

20 Next, we demonstrate simulation techniques with random-number generation. We simulate a popular casino dice game that uses most of the C++ capabilities we've presented so far. In the game, we show how to declare constants in scoped `enum`s and discuss C++20's new `using enum` declarations for accessing scoped `enum` constants directly without their type name.

We then present C++'s scope rules for determining where identifiers can be referenced in a program. We discuss features that help improve program performance, including inline functions that can eliminate a function call's overhead and reference parameters for efficiently passing large data items to functions.

Many of the applications you develop will have more than one function of the same name. You'll use this "function overloading" technique to implement functions that perform similar tasks for arguments of different types or different numbers of arguments. We introduce function templates, which define families of overloaded functions. We also demonstrate recursive functions that call themselves, directly or indirectly, through another function. Cujuumt, ul znkfehdf jsy lagqynb-ovrbozi mljapvao thqt w wjtz qarcv aj wazkrvdqxbu (see Section 5.20).

5.2 C++ Program Components

You typically write C++ programs by combining

- prepackaged functions and classes available in the C++ standard library,
- functions and classes available in a vast array of open-source and proprietary third-party libraries, and
- new functions and classes you and your colleagues write.

The C++ standard library provides a rich collection of functions and classes for math, string processing, regular expressions, input/output, file processing, dates, times, containers (collections of data), algorithms for manipulating the contents of containers, memory management, concurrent programming, asynchronous programming and more.

Functions and classes allow you to separate a program's tasks into self-contained units. You've used a combination of C++ standard library features, open-source library features and the `main` function in every program so far. In this chapter, you'll begin defining custom functions and starting in Chapter 9, you'll define custom classes.

Some motivations for using functions and classes to create program components include:

- Software reuse. For example, in earlier programs, we did not have to define how to create and manipulate `string`s or read a line of text from the keyboard—C++ provides these capabilities via the `<string>` header's `string` class and `getline` function.
- Avoiding code repetition.
- Dividing programs into meaningful functions and classes makes programs easier to test, debug and maintain.

20 To promote reusability, every function should perform a single, well-defined task, and the function's name should express that task effectively. We'll say lots more about software reusability in our treatment of object-oriented programming. C++20 introduces another construct called **modules**, which we will discuss in Chapter 16.

5.3 Math Library Functions

In the Objects-Natural case study sections, you've created objects of interesting classes, then called their member functions to perform useful tasks. Functions like `main` that are not member functions are called **global functions**.

The `<cmath>` header provides many global functions (in the `std` namespace) for common mathematical calculations. For example,

```
sqrt(900.0)
```

calculates the square root of `900.0` and returns the result, `30.0`. Function `sqrt` takes a `double` argument and returns a `double` result. There's no need to create any objects before calling function `sqrt`. Each function is called simply by specifying the function name followed by parentheses containing the arguments. Some popular math library functions are summarized in the following table. The variables `x` and `y` are of type `double`.

Function	Description	Example
`ceil(x)`	rounds x to the smallest integer not less than x	`ceil(9.2)` is `10.0`
		`ceil(-9.8)` is `-9.0`
cos(x)	trigonometric cosine of x (x in radians)	`cos(0.0)` is `1.0`
exp(x)	exponential function e^x	`exp(1.0)` is `2.718282`
		`exp(2.0)` is `7.389056`
fabs(x)	absolute value of x	`fabs(5.1)` is `5.1`
		`fabs(0.0)` is `0.0`
		`fabs(-8.76)` is `8.76`

Function	Description	Example
floor(x)	rounds x to the largest integer not greater than x	`floor(9.2)` is `9.0` `floor(-9.8)` is `-10.0`
fmod(x, y)	remainder of x/y as a floating-point number	`fmod(2.6, 1.2)` is `0.2`
log(x)	natural logarithm of x (base e)	`log(2.718282)` is `1.0` `log(7.389056)` is `2.0`
log10(x)	logarithm of x (base 10)	`log10(10.0)` is `1.0` `log10(100.0)` is `2.0`
pow(x, y)	x raised to power y (x^y)	`pow(2, 7)` is `128` `pow(9, .5)` is `3`
sin(x)	trigonometric sine of x (x in radians)	`sin(0.0)` is `0`
sqrt(x)	square root of x (where x is a non-negative value)	sqrt(9.0) is `3.0`
tan(x)	trigonometric tangent of x (x in radians)	`tan(0.0)` is `0`

C++11 Additional Math Functions

11 17 C++11 added dozens of new math functions to the `<cmath>` header. Some were entirely new, and some were other versions of existing functions but for arguments of type `float` or `long double`, rather than `double`. The two-argument **`hypot`** function, for example, calculates a right triangle's hypotenuse. C++17 added a three-argument version of `hypot` to calculate the hypotenuse in three-dimensional space. For a complete list of all the `<cmath>` header's functions, see

```
https://en.cppreference.com/w/cpp/numeric/math
```

or the C++ standard section, "Mathematical functions for floating-point types":

https://timsong-cpp.github.io/cppwp/n4861/c.math

20 C++20 New Mathematical Constants and the <numbers> Header

Before C++20, C++ did not provide common mathematical constants. Some C++ implementations defined M_PI (for π) and M_E (for *e*) and other mathematical constants via preprocessor macros.[1] When the preprocessor executes, it replaces these macro names with double floating-point values. Unfortunately, these preprocessor macros were not present in every C++ implementation. C++20's new <numbers> header[2] standardizes the following mathematical constants commonly used in many scientific and engineering applications:

---

1. We discuss the preprocessor and macros in online Appendix D.

2. Lev Minkovsky and John McFarlane, "Math Constants," July 17, 2019. Accessed December 28, 2021. http://wg21.link/p0631r8.

Constant	Mathematical expression
numbers::e	e
numbers::log2e	$\log_2 e$
numbers::log10e	$\log_{10} e$
numbers::ln2	$\log_e(2)$
numbers::ln10	$\log_e(10)$
numbers::pi	π
numbers::inv_pi	π1π
numbers::inv_sqrtpi	π1π
numbers::sqrt2	2
numbers::sqrt3	3
numbers::inv_sqrt3	13
numbers::egamma	Euler-Mascheroni γ constant
numbers::phi	(1+5)2

C++17 Mathematical Special Functions

17 C++17 added scores of **mathematical special functions** for the engineering and scientific communities to the `<cmath>` header.[3] You can see the complete list and brief examples of each on cppreference.com.[4] Each function in the following table has versions for `float`, `double` and `long double` arguments:

3. Walter E. Brown, Axel Naumann and Edward Smith-Rowland, "Mathematical Special Functions for C++17, v5," February 29, 2016. Accessed December 27, 2021. http://wg21.link/p0226r1.
4. "Mathematical Special Functions." Accessed December 27, 2021. https://en.cppreference.com/w/cpp/numeric/special_functions.

C++ 17 mathematical special functions	
associated Laguerre polynomials	irregular modified cylindrical Bessel functions
associated Legendre polynomials	cylindrical Neumann functions
beta function	exponential integral
(complete) elliptic integral of the first kind	Hermite polynomials
(incomplete) elliptic integral of the first kind	Legendre polynomials
(complete) elliptic integral of the second kind	Laguerre polynomials
(incomplete) elliptic integral of the second kind	Riemann zeta function
(complete) elliptic integral of the third kind	spherical Bessel functions (of the first kind)
(incomplete) elliptic integral of the third kind	spherical associated Legendre functions
regular modified cylindrical Bessel functions	spherical Neumann functions

C++ 17 mathematical special functions
cylindrical Bessel functions (of the first kind)

5.4 Function Definitions and Function Prototypes

Let's create a user-defined function called `maximum` that returns the largest of its three `int` arguments. When Fig. 5.1 executes, `main` reads three integers from the user. Then, line 16 calls `maximum`, which is defined in lines 20–34. In line 33, function `maximum` returns the largest value back to its caller—in this case, line 16 displays the return value.

```
 1  // fig05_01.cpp
 2  // maximum function with a function prototype.
 3  #include <iostream>
 4  #include <iomanip>
 5  using namespace std;
 6
 7  int maximum(int x, int y, int z); // function prototype
 8
 9  int main() {
10     cout << "Enter three integer values: ";
11     int int1, int2, int3;
12     cin >> int1 >> int2 >> int3;
13
14     // invoke maximum
15     cout << "The maximum integer value is: "
16        << maximum(int1, int2, int3) << '\n';
17  }
18
19  // returns the largest of three integers
20  int maximum(int x, int y, int z) {
21     int maximumValue{x}; // assume x is the largest to start
22
23     // determine whether y is greater than maximumValue
24     if (y > maximumValue) {
25        maximumValue = y; // make y the new maximumValue
26     }
27
28     // determine whether z is greater than maximumValue
29     if (z > maximumValue) {
30        maximumValue = z; // make z the new maximumValue
31     }
32
33     return maximumValue
34  }
```

```
Enter three integer grades: 86 67 75
The maximum integer value is: 86
```

```
Enter three integer grades: 67 86 75
The maximum integer value is: 86
Enter three integer grades: 67 75 86
The maximum integer value is: 86
```

Fig. 5.1 `maximum` function with a function prototype.

Function definition for `maximum`

Lines 20-31 represent the function's definition. The function's first line is also known as the function's **header**. A function definition's first line specifies its return type, function name and parentheses containing the **parameter list**, which specifies any additional information the function needs to perform its task. A parameter list may contain zero or more **parameters**, each declared with a type and a name. Two or more parameters are specified using a comma-separated list. Function `maximum` has three `int` parameters named `x`, `y` and `z`. When you call a function, each parameter receives the corresponding argument's value from the function call.

Function `maximum` first assumes that parameter `x` has the largest value, so line 21 initializes `maximumValue` to `x`'s value. Of course, parameter `y` or `z` might contain the largest value, so we compare each to `maximumValue`. Lines 24–26 determine whether `y` is greater than `maximumValue` and, if so, assign `y` to `maximumValue`. Lines 29–31 determine whether `z` is greater than `maximumValue` and, if so, assign `z` to `maximumValue`. Now, `maximumValue` contains the largest value, so line 33 returns a copy of that value to the caller.

Function Prototype for `maximum`

You must either define a function before using it or declare it, as in line 7:

```
int maximum(int x, int y, int z); // function prototype
```

This **function prototype** describes the interface to the `maximum` function without revealing its implementation. A function prototype tells the compiler the function's name, its return type and the types of its parameters, and ends with a required semicolon (;). Line 7 indicates that `maximum` returns an `int` and requires three `int` parameters to perform its task. The types in the function prototype must be the same as those in the corresponding function definition's header (line 20).

Parameter Names in Function Prototypes

The function prototype's parameter names in function prototypes are optional (the compiler ignores them), but it's recommended that you use these names for documentation purposes.

What the Compiler Does with `maximum`'s Function Prototype

When compiling the program, the compiler uses the prototype to:

- Check that `maximum`'s header (line 20) matches its prototype (line 7).
- Check that the call to `maximum` (line 16) contains the correct number and types of arguments, and that the types of the arguments are in the correct order (in this case, all the arguments are of the same type).
- Check that the value returned by the function can be used correctly in the expression that called the function. For example, a function declared with the `void` **return type** does not return a value, so it cannot be called where a value is expected, such as on the right side of an assignment or in a `cout` statement.
- Check that each argument is consistent with the type of the corresponding parameter—for example, a parameter of type `double` can receive values like 7.35, 22 or –0.03456, but not a string

like `"hello"`. If the arguments passed to a function do not match the types specified in the function's prototype, the compiler attempts to convert the arguments to those types. Section 5.6 discusses this conversion process and what happens if the conversion is not allowed.

Err Compilation errors occur if the function prototype, header and calls do not agree in the number, type and order of arguments and parameters, and the return type.

Returning Control from a Function to Its Caller

When a program calls a function, the function performs its task, then returns control (and possibly a value) to the point where the function was called. In a function that does not return a result (i.e., it has a `void` return type), control returns when the program reaches the function-ending right brace. A `void`-return-type function can explicitly return control (and no result) to the caller by executing

```
return;
```

anywhere in the function's body.

5.5 Order of Evaluation of a Function's Arguments

The commas in line 16 of Fig. 5.1 that separate function `maximum`'s arguments are not comma operators. The comma operator guarantees that its operands evaluate left-to-right. However, the order of evaluation of a function's arguments is not specified by the C++ standard. Thus, different compilers can evaluate function arguments in different orders.

Sometimes when a function's arguments are expressions, such as those with calls to other functions, the order in which the compiler evaluates the arguments could affect the values of one or more of the arguments. If the evaluation order changes between compilers, the argument values passed to the function could vary, causing subtle logic errors.

If you have doubts about the order of evaluation of a function's arguments and whether the order would affect the values passed to the function, assign the arguments to variables before the call, then pass those variables as arguments to the function.

5.6 Function-Prototype and Argument-Coercion Notes

Err A function prototype is required unless the function is defined before it's used. If a function is defined before it's called, its definition also serves as the function's prototype, so a separate prototype is unnecessary. A compilation error occurs if a function is called before it's defined and that function does not have a function prototype.

When you use a standard library function like `sqrt`, you do not have access to its definition, so it cannot be defined in your code before you call the function. Instead, you must include the header (in this case, `<cmath>`) containing the function's prototype. Even though it's possible to omit function prototypes when functions are defined before they're used, providing them avoids tying your code to the order in which functions are defined, which can easily change as a program evolves.

5.6.1 Function Signatures and Function Prototypes

A function's name and its parameter types together are known as the **function signature** or simply the **signature**. The function's return type is not part of the function signature. A function's scope is the region of a program in which the function is known and accessible. Functions in the same scope must have unique signatures. We'll say more about scope in Section 5.10.

In Fig. 5.1, if the function prototype in line 7 had been written

```
void maximum(int x, int y, int z);
```

Err the compiler would report an error because the prototype's `void` return type would differ from the function header's `int` return type. Similarly, such a prototype would cause the statement

```
cout << maximum(6, 7, 0);
```

Err to generate a compilation error because that statement depends on `maximum` returning a value to be displayed. Function prototypes help you many find errors at compile-time, which is always better than finding them at runtime.

5.6.2 Argument Coercion

An important feature of function prototypes is **argument coercion**—that is, forcing arguments to the appropriate types specified by the parameter declarations. For example, a program can call a function with an integer argument, even though the function prototype specifies a `double` parameter. The function will still work correctly, provided this is not a narrowing conversion (discussed in Section 3.8.3). A compilation error occurs if the arguments in a function call cannot be implicitly converted to the expected types specified in the function's prototype.

5.6.3 Argument-Promotion Rules and Implicit Conversions

Sometimes, argument values that do not correspond precisely to the parameter types in the function prototype can be converted by the compiler to the proper type before the function is called. These conversions occur as specified by C++'s **promotion rules**, which indicate the implicit conversions allowed between fundamental types.[5] An `int` can be converted to a `double`. A `double` also can be converted to an `int`, but this narrowing conversion truncates the `double`'s fractional part.[6] Keep in mind that `double` variables can hold numbers of much greater magnitude than `int` variables, so the loss of data in a narrowing conversion can be considerable.

5. There are additional promotion and conversion rules beyond what we discuss here. See Sections 7.3 and 7.4 of the C++ standard for more information. https://timsong-cpp.github.io/cppwp/n4861/conv and https://timsong-cpp.github.io/cppwp/n4861/expr.arith.conv.

11 6. Recall from Section 3.8.3 that C++11 braced initializers do not allow narrowing conversions.

CG Values might also be modified when converting large integer types to small integer types (e.g., `long` to `short`), signed to unsigned types, or unsigned to signed types. Variables of **`unsigned`** integer types can represent values from 0 to approximately twice the positive range of the corresponding signed integer types. The `unsigned` types are used primarily for bit manipulation (online Appendix E). The C++ Core Guidelines indicate that `unsigned` types should not be used to ensure or document that a value is nonnegative.[7]

7. C++ Core Guidelines, "ES.106: Don't Try to Avoid Negative Values By Using `unsigned`." Accessed December 25, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-nonnegative.

The promotion rules also apply to **mixed-type expressions** containing values of two or more data types. Each value's type is promoted to the expression's "highest" type. The promotion uses a temporary copy of each value—the original values remain unchanged. The following table lists the arithmetic types in order from "highest type" to "lowest type":

Data types	
`long double`	
`double`	
`float`	
`unsigned long long int`	(synonymous with `unsigned long long`)
`long long int`	(synonymous with `long long`)
`unsigned long int`	(synonymous with `unsigned long`)
`long int`	(synonymous with `long`)
`unsigned int`	(synonymous with `unsigned`)
`int`	
`unsigned short int`	(synonymous with `unsigned short`)
`short int`	(synonymous with `short`)
`unsigned char`	
`char` and `signed char`	
`bool`	

Conversions Can Result in Incorrect Values

Converting values to lower types can cause narrowing conversion errors or warnings. If you pass a `double` argument to a `square` function with an `int` parameter, the argument is converted to `int` (a lower type and thus a narrowing conversion), and `square` could return an incorrect value. For example, `square(4.5)` would return `16`, not `20.25`. Some compilers warn you about this. For example, Microsoft Visual C++ issues the warning,

```
'argument': conversion from 'double' to 'int', possible loss of data
```

Narrowing Conversions with the Guidelines Support Library

CG If you must perform an explicit narrowing conversion, the C++ Core Guidelines recommend using **`narrow_cast`**[8] from the **Guidelines Support Library (GSL)**. This library has several implementations. Microsoft's open-source version has been tested on numerous platform/compiler combinations, including our three preferred compilers and platforms. You can download the GSL from

8. C++ Core Guidelines. Accessed May 10, 2020. http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-narrowing.

```
https://github.com/Microsoft/GSL
```

For your convenience, we provided this GSL with this book's code examples in the sub-folder `libraries/GSL`.

The GSL is a header-only library, so you can use it in your programs simply by including the header `<gsl/gsl>`. You must point your compiler to the `GSL` folder's `include` sub-folder, so the compiler knows where to find the header file, as you did when you used class `BigNumber` at the end of Section 3.12. The following statement uses a `narrow_cast` (from namespace `gsl`) to convert the double value `7.5` to the `int` value `7`:

```
gsl::narrow_cast<int>(7.5)
```

The value in parentheses is converted to the type in angle brackets, `<>`.

5.7 C++ Standard Library Headers

The C++ standard library is divided into many headers. Each contains function prototypes for the related functions in that header. The headers also contain definitions of various class types and functions, and constants needed by those functions. A header "instructs" the compiler on how to interface with library and user-written components.

The following table lists some common C++ standard library headers, many of which are discussed later in this book. The term "macro" in this table is discussed in detail in online Appendix D, Preprocessor. For a complete list of the 96 C++20 standard library headers, visit

```
https://en.cppreference.com/w/cpp/header
```

On that page, you'll see approximately three dozen additional headers that are marked as either deprecated or removed. Deprecated headers are ones you should no longer use, and removed headers are no longer included in the C++ standard library.

Standard library header	Explanation
`<iostream>`	Function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and covered in more detail in online Chapter 18, Stream I/O and C++20 Text Formatting.
`<iomanip>`	Function prototypes for stream manipulators that format streams of data. This header is first used in Section 3.7 and is discussed in more detail in online Chapter 18.
`<cmath>`	Function prototypes for math library functions (Section 5.3).
`<cstdlib>`	Function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 5.8; Chapter 11, Operator Overloading, Copy/Move Semantics and Smart Pointers; and Chapter 12, Exceptions and a Look Forward to Contracts.
11 `<random>`	C++11's random-number generation capabilities (discussed in this chapter and used in subsequent chapters).
11 20 `<ctime>`, `<chrono>`	Function prototypes and types for manipulating the time and date. `<ctime>` is used in Section 5.8. `<chrono>` was introduced in C++11 and enhanced with many more features in C++20. We use several `<chrono>` timing features in **Chapter 17, Parallel Algorithms and Concurrency: A High-Level View**.
11 `<array>`, `<vector>`, `<list>`, `<tuple>`, `<forward_list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<unordered_map>`, `<unordered_set>`, `<set>`, `<bitset>`	These headers contain classes that implement the C++ standard library containers. Containers are standard implementations of commonly used data structures. The `<array>` and `<vector>` headers are first introduced in Chapter 6, `array`s, `vector`s, Ranges and Functional-Style Programming. We discuss all these headers in Chapter 13, Standard Library Containers and Iterators. `<array>`, `<forward_list>`, `<tuple>`, `<unordered_map>` and `<unordered_set>` were introduced in C++11.

Standard library header	Explanation
`<cctype>`	Function prototypes for functions that test characters for certain properties (such as whether the character is a digit or punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters, and vice versa.
`<cstring>`	Function prototypes for C-style string-processing functions.
`<typeinfo>`	Classes for runtime type identification (determining data types at execution time).
`<exception>`, `<stdexcept>`	Classes for exception handling (discussed in Chapter 12).
`<memory>`	Classes and functions for managing memory allocation. This header is used in Chapter 12.
`<fstream>`	Function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 8, `string`s, `string_view`s, Text Files, CSV Files and Regex).
`<string>`	Definition of class `string` and functions `getline` and `to_string` (discussed in Chapter 8).
`<sstream>`	Function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 8).
`<functional>`	Classes and functions used by C++ standard library algorithms. This header is used in Chapter 14, Standard Library Algorithms and C++20 Ranges & Views.
`<iterator>`	Classes for accessing data in the C++ standard library containers. This header is used in Chapter 13, Standard Library Containers and Iterators.
`<algorithm>`	Functions for manipulating data in C++ standard library containers. This header is used in Chapter 13.

Standard library header	Explanation
`<cassert>`	Macros for adding diagnostics that aid program debugging. This header is used in online Appendix D, Preprocessor.
`<cfloat>`	Floating-point size limits of the system.
`<climits>`	Integral size limits of the system.
`<cstdio>`	Function prototypes for C-style standard input/output library functions.
`<locale>`	Classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation).
`<limits>`	Classes for defining the numerical data type limits on each computer platform—this is C++'s version of `<climits>` and `<cfloat>`.
`<utility>`	Classes and functions that are used by many C++ standard library headers.
11 14 `<thread>`, `<mutex>`, `<shared_mutex>`, `<future>`, `<condition_variable>`	Capabilities added in C++11 and C++14 for multithreaded application development so your applications can take advantage of multi-core processors (discussed in Chapter 17, Parallel Algorithms and Concurrency: A High-Level View).
17 Some Key C++17 New Headers	
`<any>`	A class for holding a value of any copyable type.
`<optional>`	A template to represent an object that may or may not have a value (discussed in Chapter 13).
`<variant>`	Features used to create and manipulate objects of a specified set of types (discussed in Chapter 10, OOP: Inheritance and Runtime Polymorphism).
`<execution>`	Features used with the Standard Template Library's parallel algorithms (discussed in Chapter 17).

Standard library header	Explanation
`<filesystem>`	Capabilities for interacting with the local file system's files and folders.
20 Some Key C++20 New Headers	
`<concepts>`	Capabilities for constraining the types that can be used with templates (discussed in Chapter 15, Templates, C++20 Concepts and Metaprogramming).
`<coroutine>`	Capabilities for asynchronous programming with coroutines (discussed in Chapter 17).
`<compare>`	Support for the new three-way comparison operator `<=>` (discussed in Chapter 11).
`<format>`	New concise and powerful text-formatting capabilities (discussed throughout the book).
`<ranges>`	Capabilities that support functional-style programming (discussed in Chapter 6 and Chapter 13).
`<span>`	Capabilities for creating views into contiguous sequences of objects (discussed in Chapter 17.
`<bit>`	Standardized bit-manipulation operations.
`<stop_token>`, `<semaphore>`, `<latch>`, `<barrier>`	Additional capabilities that support the multithreaded application-development features added in C++11 and C++14 (discussed in Chapter 17).

5.8 Case Study: Random-Number Generation

We now take a brief and hopefully entertaining diversion into a popular programming application—simulation and game playing. In this section and the next, we develop a game-playing program that includes multiple functions. The program outputs for this section's examples were produced using Visual Studio 2022 Community Edition. Your outputs may vary, based on your compiler and platform.

Header `<random>`

11 Sec The element of chance can be introduced into your applications with features from C++11's **`<random>` header**. These features replace the deprecated `rand` function, which was inherited into C++ from the C standard library. The `rand` function does not have "good statistical properties" and can be predictable.[9] This makes programs using `rand` less secure.

C++11 provides a more secure library of random-number capabilities that can produce **nondeterministic random numbers**—a set of random numbers that can't be predicted. Such random-number generators are used in simulations and security scenarios where predictability is undesirable.

Random-number generation is a sophisticated topic for which mathematicians have developed many algorithms with different statistical properties. For flexibility based on how random numbers are used in programs, C++11 provides many classes that represent various **random-number generation engines** and **distributions**:

- An **engine** implements a random-number generation algorithm that produces random numbers.
- A **distribution** controls the range of values produced by an engine, the value's types (e.g., `int`, `double`, etc.) and the value's statistical properties.

We'll use the default random-number generation engine—**`default_random_engine`**—and a **`uniform_int_distribution`**, which evenly distributes random integers over a specified range. The default range is from 0 to the maximum `int` value on your platform. If the `default_random_engine` and `uniform_int_distribution` truly produce integers at random, every number between 0 and the maximum `int` value has an equal chance (or probability) of being chosen each time the program requests a random number.

The range of values produced directly by the `default_random_engine` often differs from what a specific application requires. For example, a program that simulates coin tossing might need only 0 for "heads" and 1 for "tails." A program that simulates rolling a six-sided die would require random integers in the range 1 to 6. A program that randomly predicts the next type of spaceship (out of four possibilities) that will fly across the horizon in a video game might require random integers in the range 0 through 3. To specify the value range, you'll initialize the `uniform_int_distribution` with the starting and ending values in the range your application requires.

For a complete list of engines and distributions, visit:

```
https://en.cppreference.com/w/cpp/header/random
```

For more details on C++11 random-number generation features, see the paper, "Random Number Generation in C++11."[10]

5.8.1 Rolling a Six-Sided Die

Figure 5.2 simulates and displays ten random rolls of a six-sided die. Line 9 creates a `default_random_engine` object named `engine` to produce random numbers. Line 12 initializes the `uniform_int_distribution` object `randomDie` with `{1, 6}`, which indicates that it produces `int` values in the range 1 to 6. The expression

```
randomDie(engine)
```

```
1  // fig05_02.cpp
2  // Producing random integers in the range 1 through 6.
3  #include <iostream>
4  #include <random> // contains C++11 random-number generation features
5  using namespace std;
6
7  int main() {
8     // engine that produces random numbers
9     default_random_engine engine{};
10
11    // distribution that produces the int values 1-6 with equal likelihood
12    uniform_int_distribution randomDie{1, 6};
13
14    // display 10 random die rolls
15    for (int counter{1}; counter <= 10; ++counter) {
16       cout << randomDie(engine) << " ";
17    }
18
19    cout << '\n';
20 }
```

```
3 1 3 6 5 2 6 6 1 2
```

Fig. 5.2 Producing random integers in the range 1 through 6.

in line 16 returns one random `int` in the range 1 to 6.

9. Fred Long, "Do Not Use the `rand()` Function for Generating Pseudorandom Numbers." Last modified by Jill Britton on November 20, 2021. Accessed December 27, 2021. https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers.

10. Walter E. Brown, "Random Number Generation in C++11," March 12, 2013. Accessed December 27, 2021. https://isocpp.org/files/papers/n3551.pdf.

5.8.2 Rolling a Six-Sided Die 60,000,000 Times

To show that the random values produced in the preceding program occur with approximately equal likelihood, Fig. 5.3 simulates 60,000,000 rolls of a die.[11] Each `int` in the range 1 to 6 should appear approximately 10,000,000 times (one-sixth of the rolls). The program's output confirms this.
The `face` variable's definition in the `switch`'s initializer (line 23) is preceded by `const`. This is a good practice for any variable that should not change once initialized. This enables the compiler to report errors if you accidentally modify the variable.

11. When co-author Harvey Deitel first implemented this example for his classes in 1976, he performed only 600 die rolls—6000 would have taken too long. On our system, this program took approximately five seconds to complete 60,000,000 die rolls! 600,000,000 die rolls took approximately one minute. The die rolls occur sequentially. In our concurrency chapter, we'll explore how to parallelize applications to take advantage of today's multi-core computers.

```
1  // fig05_03.cpp
2  // Rolling a six-sided die 60,000,000 times.
```

```cpp
#include <fmt/format.h>
#include <iostream>
#include <random>
using namespace std;

int main() {
   // set up random-number generation
   default_random_engine engine{};
   uniform_int_distribution randomDie{1, 6};

   int frequency1{0}; // count of 1s rolled
   int frequency2{0}; // count of 2s rolled
   int frequency3{0}; // count of 3s rolled
   int frequency4{0}; // count of 4s rolled
   int frequency5{0}; // count of 5s rolled
   int frequency6{0}; // count of 6s rolled

   // summarize results of 60,000,000 rolls of a die
   for (int roll{1}; roll <= 60'000'000; ++roll) {
      // determine roll value 1-6 and increment appropriate counter
      switch (const int face{randomDie(engine)}) {
         case 1:
            ++frequency1; // increment the 1s counter
            break;
         case 2:
            ++frequency2; // increment the 2s counter
            break;
         case 3:
            ++frequency3; // increment the 3s counter
            break;
         case 4:
            ++frequency4; // increment the 4s counter
            break;
         case 5:
            ++frequency5; // increment the 5s counter
            break;
         case 6:
            ++frequency6; // increment the 6s counter
            break;
         default: // invalid value
            cout << "Program should never get here!";
            break;
      }
   }

   cout << fmt::format("{:>4}{:>13}\n", "Face", "Frequency"); // headers
   cout << fmt::format("{:>4d}{:>13d}\n", 1, frequency1)
      << fmt::format("{:>4d}{:>13d}\n", 2, frequency2)
      << fmt::format("{:>4d}{:>13d}\n", 3, frequency3)
      << fmt::format("{:>4d}{:>13d}\n", 4, frequency4)
      << fmt::format("{:>4d}{:>13d}\n", 5, frequency5)
      << fmt::format("{:>4d}{:>13d}\n", 6, frequency6);
}
```

```
Face Frequency
   1   9997896
```

```
2  10000608
3   9996800
4  10000729
5  10003444
6  10000523
```

Fig. 5.3 Rolling a six-sided die 60,000,000 times.

The `switch`'s default case (lines 42–44) should never execute because the controlling expression (`face`) always has values in the range 1–6. Many programmers provide a `default` case in every `switch` statement to catch errors, even if they feel confident that their programs are error-free. After introducing arrays in Chapter 6, we show how to elegantly replace the entire `switch` in Fig. 5.3 with a single-line statement.

5.8.3 Seeding the Random-Number Generator

Executing the program of Fig. 5.2 again produces

```
3 1 3 6 5 2 6 6 1 2
```

This is the same sequence of values shown in Fig. 5.2. How can these be random numbers?

The `default_random_engine` actually generates **pseudorandom numbers**. Repeatedly executing the programs of Figs. 5.2 and 5.3 produces sequences of numbers that appear to be random. However, the sequences actually repeat themselves each time these programs execute. When debugging a simulation, random-number repeatability is essential for proving that corrections to the program work properly.

Once you've thoroughly debugged your simulation, you can condition it to produce a different sequence of random numbers for each execution. This is called **randomizing**. You initialize the `default_random_engine` with an `unsigned int` argument that **seeds** the `default_random_engine` to produce a different sequence of random numbers for each execution.

Seeding the `default_random_engine`

Sec Figure 5.4 demonstrates seeding the `default_random_engine` (line 15) with an `unsigned int` that you input (line 12). The program produces a different random-number sequence each time it executes, **provided that you enter a different seed**. We used the same seed in the first and third sample outputs, so the same series of 10 numbers is displayed in each. For security, you must ensure that your program seeds the random-number generator differently (and only once) each time the program executes; otherwise, an attacker could determine the sequence of pseudorandom numbers that would be produced.

```
1  // fig05_04.cpp
2  // Randomizing the die-rolling program.
3  #include <iostream>
4  #include <iomanip>
5  #include <random>
```

```
 6   using namespace std;
 7
 8   int main() {
 9      unsigned int seed{0}; // stores the seed entered by the user
10
11      cout << "Enter seed: ";
12      cin >> seed;
13
14      // set up random-number generation
15      default_random_engine engine{seed}; // seed the engine
16      uniform_int_distribution randomDie{1, 6};
17
18      // display 10 random die rolls
19      for (int counter{1}; counter <= 10; ++counter) {
20         cout << randomDie(engine) << " ";
21      }
22
23      cout << '\n';
24   }
```

```
Enter seed: 67
6 2 5 6 6 2 2 6 2 1
Enter seed: 432
3 5 6 1 6 1 4 4 2 2
Enter seed: 67
6 2 5 6 6 2 2 6 2 1
```

Fig. 5.4 Randomizing the die-rolling program.

5.8.4 Seeding the Random-Number Generator with `random_device`

11 Sec Perf The proper way to randomize is to seed the random-number generator engine with a C++11 **`random_device`** object (from header `<random>`), as we'll do in Fig. 5.5.
A `random_device` produces uniformly distributed, **nondeterministic random integers**, which cannot be predicted.[12] However, the `random_device` documentation indicates that for performance reasons, it's typically used only to seed random-number generation engines.[13] **Also, `random_device` might be deterministic on some platforms.[14] So, be sure to check your compiler's documentation before relying on this for secure applications. For example, Visual C++'s implementation provides nondeterministic, cryptographically secure random numbers.[15]**

12. "Nondeterministic Algorithm." Wikipedia. Wikimedia Foundation. Accessed May 2, 2021. https://en.wikipedia.org/wiki/Nondeterministic_algorithm.
13. "`std::random_device`." Accessed December 27, 2021. https://en.cppreference.com/w/cpp/numeric/random/random_device.
14. "`std::random_device`." Accessed December 27, 2021. https://en.cppreference.com/w/cpp/numeric/random/random_device.
15. "`random_device` Class," August 3, 2021. Accessed December 27, 2021. https://docs.micro-soft.com/en-us/cpp/standard-library/random-device-class.

5.9 Case Study: Game of Chance; Introducing Scoped **`enum`**s

One of the most popular games of chance is a dice game known as "craps," which is played in casinos and back alleys worldwide. The rules of the game are straightforward:

A player rolls two dice. Each die has six faces containing 1, 2, 3, 4, 5 and 6 spots. After the dice have come to rest, the sum of the spots on the two upward faces is calculated. If the sum is 7 or 11 on the first roll, the player wins. If the sum is 2, 3 or 12 on the first roll (called "craps"), the player loses (i.e., the "house" wins). If the sum is 4, 5, 6, 8, 9 or 10 on the first roll, then that sum becomes the player's "point." To win, you must keep rolling the dice until you "make your point." The player loses by rolling a 7 before making the point.

In the rules, notice that the player must roll two dice on the first roll and all subsequent rolls. We will define a `rollDice` function to roll the dice and compute and display their sum. The function may be called multiple times—once for the game's first roll and possibly many more times if the player does not win or lose on the first roll. Below are the outputs of several sample executions showing:

- winning on the first roll by rolling a 7,
- winning on a subsequent roll by "making the point" before rolling a 7,
- losing on the first roll by rolling a 12, and
- losing on a subsequent roll by rolling a 7 before "making the point."

```
Player rolled 2 + 5 = 7
Player wins
Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins
Player rolled 6 + 6 = 12
Player loses
```

```
Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses
```

Implementing the Game

The craps program (Fig. 5.5) simulates the game using two functions—`main` and `roll-Dice`—and the `switch`, `while`, `if`...`else` and nested `if`...`else` statements. Function `rollDice`'s prototype (line 8) indicates that the function takes no arguments (empty parentheses) and returns an `int` (the sum of the dice).

```
1   // fig05_05.cpp
2   // Craps simulation.
```

```
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <random>
6   using namespace std;
7
8   int rollDice(); // rolls dice, calculates and displays sum
```

Fig. 5.5 Craps simulation.

11 C++11 Scoped `enum`s

The player may win or lose on the first roll or any subsequent roll. The program tracks this with the variable `gameStatus`, which line 15 declares to be of the new type `Status`. Line 12 declares a user-defined type called a **scoped enumeration** and is introduced by the keywords **`enum class`**, followed by a type name (`Status`) and a set of identifiers representing integer constants.

```
10  int main() {
11     // scoped enumeration with constants that represent the game status
12     enum class Status {keepRolling, won, lost};
13
14     int myPoint{0}; // point if no win or loss on first roll
15     Status gameStatus{Status::keepRolling}; // game is not over
```

The underlying values of these **enumeration constants** are of type `int`, start at `0` and increment by `1`, by default. In the `Status` enumeration, the constant `keepRolling` has the value 0, `won` has the value 1, and `lost` has the value 2. The identifiers in an `enum class` must be unique, but multiple identifiers may have the same value. Variables of user-defined type `Status` can be assigned only the constants declared in the enumeration.

CG By convention, you should capitalize the first letter of an `enum class`'s name and the first letter of each subsequent word in a multi-word `enum class` name (e.g., `ProductCode`). The C++ Core Guidelines state that constants in an `enum class` should use the same naming conventions as variables.[16,17]

16. C++ Core Guidelines. Accessed May 11, 2020. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-caps.

17. In legacy C++ code, you'll commonly see `enum` constants in all uppercase letters—that practice is now deprecated.

To reference a scoped `enum` constant, qualify the constant with the scoped `enum`'s type name (i.e., `Status`) and the scope-resolution operator (`::`), as shown in line 15, which initializes `gameStatus` to `Status::keepRolling`. For a win, the program sets `gameStatus` to `Status::won`. For a loss, the program sets `gameStatus` to `Status::lost`.

Winning or Losing on the First Roll

The following `switch` determines whether the player wins or loses on the first roll.

```
17     // determine game status and point (if needed) based on first roll
18     switch (const int sumOfDice{rollDice()}) {
19        case 7: // win with 7 on first roll
20        case 11: // win with 11 on first roll
21           gameStatus = Status::won;
22           break;
```

```
23          case 2: // lose with 2 on first roll
24          case 3: // lose with 3 on first roll
25          case 12: // lose with 12 on first roll
26             gameStatus = Status::lost;
27             break;
28          default: // did not win or lose, so remember point
29             myPoint = sumOfDice; // remember the point
30             cout << fmt::format("Point is {}\n", myPoint);
31             break; // optional (but recommended) at end of switch
32       }
```

The `switch`'s initializer (line 18) creates the variable `sumOfDice` and initializes it by calling `rollDice`. If the roll is `7` or `11`, line 21 sets `gameStatus` to `Status::won`. If the roll is `2`, `3`, or `12`, line 26 sets `gameStatus` to `Status::lost`. For other values, `gameStatus` remains unchanged (`Status::keepRolling`), line 29 saves `sumOfDice` in `myPoint`, and line 30 displays `myPoint`.

Continuing to Roll

After the first roll, if `gameStatus` is `Status::keepRolling`, execution proceeds with the following `while` statement.

```
34       // while game is not complete
35       while (Status::keepRolling == gameStatus) { // not won or lost
36          // roll dice again and determine game status
37          if (const int sumOfDice{rollDice()}; sumOfDice == myPoint) {
38             gameStatus = Status::won;
39          }
40          else if (sumOfDice == 7) { // lose by rolling 7 before point
41             gameStatus = Status::lost;
42          }
43       }
```

17 In each loop iteration, the `if` statement's initializer (line 37) calls `rollDice` to produce a new `sumOfDice`. If `sumOfDice` matches `myPoint`, line 38 sets `gameStatus` to `Status::won`, and the loop terminates. If `sumOfDice` is `7`, the program sets `gameStatus` to `Status::lost` (line 41), and the loop terminates. Otherwise, the loop continues executing.

Displaying Whether the Player Won or Lost

When the preceding loop terminates, the program proceeds to the following `if`...`else` statement, which prints `"Player wins"` if `gameStatus` is `Status::won` or `"Player loses"` if `gameStatus` is `Status::lost`.

```
45       // display won or lost message
46       if (Status::won == gameStatus) {
47          cout << "Player wins\n";
48       }
49       else {
50          cout << "Player loses\n";
51       }
52    }
```

Function `rollDice`

Function `rollDice` rolls two dice (lines 61–62), calculates their sum (line 63), prints their faces and sum (line 66), and returns the sum (line 68).

```
54   // roll dice, calculate sum and display results
55   int rollDice() {
56      // set up random-number generation
57      static random_device rd; // used to seed the default_random_engine
58      static default_random_engine engine{rd()}; // rd() produces a seed
59      static uniform_int_distribution randomDie{1, 6};
60
61      const int die1{randomDie(engine)}; // first die roll
62      const int die2{randomDie(engine)}; // second die roll
63      const int sum{die1 + die2}; // compute sum of die values
64
65      // display results of this roll
66      cout << fmt::format("Player rolled {} + {} = {}\n", die1, die2, sum);
67
68      return sum;
69   }
```

Generally, each program that uses random numbers creates one random-number generator engine, seeds it, then uses it throughout the program. In this program, only function `rollDice` needs access to the random-number generator, so lines 57–59 define the `random_device`, `default_random_engine` and `uniform_int_distribution` objects as **`static` local variables**. Unlike other local variables, which exist only until a function call terminates, `static` local variables retain their values between function calls. Declaring the objects in lines 57–59 `static` ensures they are created only the first time `rollDice` is called. They are then reused in all subsequent `rollDice` calls. The expression `rd()` in line 58 gets a nondeterministic random integer from the `random_device` object and uses it to seed the `default_random_engine`, enabling the program to produce different results each time we execute it.

Additional Notes Regarding Scoped `enum`s

Qualifying an `enum class`'s constant with its type name and `::` explicitly identifies the constant as being in that type's scope. If another `enum class` contains the same identifier, it's always clear which constant is being used because the type name and `::` are required. In general, use unique `enum` constant values to help prevent hard-to-find logic errors.

Another popular scoped enumeration is

```
enum class Month {jan = 1, feb, mar, apr, may, jun, jul, aug,
   sep, oct, nov, dec};
```

which creates user-defined `enum class` type `Month` with enumeration constants representing the months of the year. The first value in the preceding enumeration is explicitly set to `1`, so the remaining values increment from `1`, resulting in the values `1` through `12`. Any constant can be assigned an integer value in the `enum class` definition. Subsequent constants have a value 1 higher than the preceding constant until the next explicit setting.

Enumeration Types Before C++11

Enumerations also can be defined with the keyword `enum` followed by a type name and a set of integer constants represented by identifiers, as in

```
enum Status {keepRolling, won, lost};
```

CG The constants in such an `enum` are unscoped—you can refer to them simply by their names `keepRolling`, `won` and `lost`. If two or more unscoped `enum`s contain constants with the same names, this can lead to naming conflicts and compilation errors. The C++ Core Guidelines recommend always using `enum class`.[18]

18. C++ Core Guidelines, “Enum.3: Prefer Class `enum`s over 'Plain' `enum`s.” Accessed December 26, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Renum-class.

C++11: Specifying the Type of an `enum`'s Constants

11 An enumeration’s constants have integer values. An unscoped `enum`’s underlying type depends on its constants’ values and is guaranteed to be large enough to store them. A scoped `enum`’s underlying integral type is `int`, but you can specify a different type by following the type name with a colon (`:`) and the integral type. For example, we can specify that the constants in the `enum class Status` should have type `short`, as in

```
enum class Status : short {keepRolling, won, lost};
```

20 C++20: `using enum` Declaration

If the type of an `enum class`’s constants is obvious based on the context in which they’re used—such as in our craps example—C++20’s **`using enum` declaration**[19,20] allows you to reference an `enum class`’s constants without the type name and scope-resolution operator (`::`). For example, adding the following statement after the `enum class` declaration

19. Gašper Ažman and Jonathan Müller, “Using Enum,” July 16, 2019. Accessed December 28, 2021. http://wg21.link/p1099r5.
20. At the time of this writing, this feature works only in Microsoft’s Visual C++ compiler.

```
using enum Status;
```

would allow the rest of the program to use `keepRolling`, `won` and `lost`, rather than `Status::keepRolling`, `Status::won` and `Status::lost`, respectively. You also may use an individual `enum class` constant with a declaration of the form

```
using Status::keepRolling;
```

This would allow your code to use `keepRolling` without the `Status::` qualifier. Generally, such `using` declarations should be placed inside the block that uses them.

5.10 Scope Rules

The portion of a program where an identifier can be used is known as its **scope**. For example, when we declare a local variable in a block, it can be referenced only

- from the point of its declaration in that block to the end of that block and
- in nested blocks that appear within that block after the variable's declaration.

This section discusses block scope and global namespace scope. Parameter names in function prototypes have **function parameter scope** and are known only in the prototype in which they appear. Later we'll see other scopes, including **class scope** in Chapter 9 and **function scope** and **namespace scope** in online Chapter 20.

Block Scope

Identifiers declared in a block have **block scope**, which begins at the identifier's declaration and ends at the block's terminating right brace (`}`). Local variables have block scope, as do function parameters. Any block can contain variable declarations. In nested blocks, if an identifier in an outer block has the same name as an identifier in an inner block, the one in the outer block is "hidden" until the inner block terminates. The inner block "sees" its own local variable's value and not that of the enclosing block's identically named variable.

Accidentally using the same name for an identifier in an inner block that's used for an identifier in an outer block when, in fact, you want the identifier in the outer block to be visible for the duration of the inner block, is typically a logic error. Avoid variable names in inner scopes that hide names in outer scopes. Most compilers will warn you about this.

As you saw, in Fig. 5.5, local variables also may be declared **`static`**. Such variables also have block scope, but unlike other local variables, a `static` local variable retains its value when the function returns to its caller. The next time the function is called, the `static` local variable contains the value it had when the function last completed execution. The following statement declares a `static` local variable `count` and initializes it to 1:

```
static int count{1};
```

By default, `static` local variables of numeric types are initialized to zero—though explicit initialization is preferred. Default initialization of non-fundamental-type variables depends on the type. For example, a `string`'s default value is the empty string (`""`). We'll say more about default initialization in later chapters.

Global Namespace Scope

An identifier declared outside any function or class has **global namespace scope**. Such an identifier is "known" to all functions after its declaration in the source-code file. Function definitions, function prototypes placed outside a function, class definitions and global variables all have global namespace scope. **Global variables** are created by placing variable declarations outside any class or function definition. Such variables retain their values throughout a program's execution.

Declaring a variable as global rather than local allows unintended **side effects** to occur when a function that does not need access to the variable accidentally or maliciously modifies it. Except for truly global resources, like `cin` and `cout`, avoid global variables. This is an example of the **principle of least privilege**, which is fundamental to good software engineering. It states that code should be granted *only* the amount of privilege and access that it needs to accomplish its designated task, but no more. An example of this is the scope of a local variable, which should not be visible when it's not needed. A local variable is created when the function is called, used by that function while it executes, then goes away when the function returns. The principle of least privilege makes your programs more robust by preventing code from accidentally (or maliciously) modifying variable values that should not be accessible to it. It also makes your programs easier to read and maintain.

In general, variables should be declared in the narrowest scope in which they need to be accessed. Variables used only in a particular function should be declared as local variables in that function rather than as global variables.

Scope Demonstration

Figure 5.6 demonstrates scoping issues with global variables, local variables and `static` local variables. We broke up this example into smaller pieces with their corresponding outputs for discussion purposes. Only the first piece has a figure caption—we'll do this for many subsequent examples throughout the book. Line 10 declares and initializes global variable `x` to 1. This global variable is hidden in any block (or function) that declares a variable named `x`.

```
1  // fig05_06.cpp
2  // Scoping example.
3  #include <iostream>
4  using namespace std;
5
6  void useLocal(); // function prototype
7  void useStaticLocal(); // function prototype
8  void useGlobal(); // function prototype
9
10 int x{1}; // global variable
```

Fig. 5.6 Scoping example.

Function `main`

In `main`, line 13 displays global variable `x`'s value. Line 15 initializes local variable `x` to `5`. Line 17 outputs this variable to show that the global `x` is hidden in `main`. Next, lines 19–23 define a new block in `main` in which another local variable `x` is initialized to `7` (line 20). Line 22 outputs this variable to show that it hides `x` in `main`'s outer block and the global `x`. When the block exits, the `x` with value `7` is destroyed automatically. Next, line 25 outputs the local variable `x` in the outer block of `main` to show that it's no longer hidden.

```
12 int main() {
13    cout << "global x in main is " << x << '\n';
14
15    const int x{5}; // local variable to main
16
17    cout << "local x in main's outer scope is " << x << '\n';
18
19    { // block starts a new scope
20       const int x{7}; // hides both x in outer scope and global x
21
22       cout << "local x in main's inner scope is " << x << '\n';
23    }
24
25    cout << "local x in main's outer scope is " << x << '\n';
```

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5
```

To demonstrate other scopes, the program defines three functions—`useLocal`, `useStaticLocal` and `useGlobal`—each of which takes no arguments and returns nothing. The rest of `main` (shown below) calls each function twice in lines 27–32. After executing functions `useLocal`, `useStaticLocal` and `useGlobal` twice each, the program prints the local variable `x` in `main` again to show that none of the function calls modified the value of `x` in `main`, because the functions all referred to variables in other scopes.

```
27      useLocal(); // useLocal has local x
28      useStaticLocal(); // useStaticLocal has static local x
29      useGlobal(); // useGlobal uses global x
30      useLocal(); // useLocal reinitializes its local x
31      useStaticLocal(); // static local x retains its prior value
32      useGlobal(); // global x also retains its prior value
33
34      cout << "\nlocal x in main is " << x << '\n';
35   }
36
```

```
local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Function `useLocal`

Function `useLocal` initializes local variable `x` to `25` (line 39). When `main` calls `useLocal` (lines 27 and 30), the function prints the variable `x`, increments it and prints it again before returning program control to its caller. Each time the program calls this function, the function recreates local variable `x` and reinitializes it to `25`.

```
37   // useLocal reinitializes local variable x during each call
38   void useLocal() {
39      int x{25}; // initialized each time useLocal is called
40
41      cout << "\nlocal x is " << x << " on entering useLocal\n";
42      ++x;
43      cout << "local x is " << x << " on exiting useLocal\n";
```

```
44    }
```

Function `useStaticLocal`

Function `useStaticLocal` declares `static` variable `x`, initializing it to `50`. Local `static` variables retain their values, even when they're out of scope (i.e., the enclosing function is not executing). When line 28 in `main` calls `useStaticLocal`, the function prints its local `x`, increments it and prints it again before the function returns program control to its caller. In the next call to this function (line 31), `static` local variable `x` contains the value `51`. The initialization in line 50 occurs only the first time `useStaticLocal` is called (line 28).

```
46    // useStaticLocal initializes static local variable x only the
47    // first time the function is called; value of x is saved
48    // between calls to this function
49    void useStaticLocal() {
50       static int x{50}; // initialized first time useStaticLocal is called
51
52       cout << "\nlocal static x is " << x
53          << " on entering useStaticLocal\n";
54       ++x;
55       cout << "local static x is " << x
56          << " on exiting useStaticLocal\n";
57    }
```

Function `useGlobal`

Function `useGlobal` does not declare any variables. So, when it refers to variable `x`, the global `x` (line 10, preceding `main`) is used. When `main` calls `useGlobal` (line 29), the function prints the global variable `x`, multiplies it by 10 and prints it again before the function returns to its caller. The next time `main` calls `useGlobal` (line 32), the global variable has its modified value, `10`.

```
59    // useGlobal modifies global variable x during each call
60    void useGlobal() {
61       cout << "\nglobal x is " << x << " on entering useGlobal\n";
62       x *= 10;
63       cout << "global x is " << x << " on exiting useGlobal\n";
64    }
```

5.11 Inline Functions

Implementing a program as a set of functions is good from a software-engineering standpoint, but function calls involve execution-time overhead. C++ provides **inline functions** to help reduce function-call overhead. Placing **`inline`** before a function's return type in the function definition advises the compiler to generate a copy of the function's body code in every place where the function is called (when appropriate) to avoid a function call. This often makes the program larger. The compiler can ignore the `inline` qualifier. Reusable `inline` functions are typically placed in headers so that their definitions can be inlined in each source file that uses them.

CG If you change an `inline` function's definition, you must recompile any code that calls that function. Though compilers can inline code for which you have not explicitly used `inline`, the C++ Core Guidelines indicate that you should declare "small and time-critical" functions `inline`.[21]

21. C++ Core Guidelines. Accessed December 28, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-inline.

The following site provides an extensive FAQ on the subtleties of `inline` functions:

```
https://isocpp.org/wiki/faq/inline-functions
```

Figure 5.7 uses `inline` function `cube` (lines 9–11) to calculate the volume of a cube.

```
1  // fig05_07.cpp
2  // inline function that calculates the volume of a cube.
3  #include <iostream>
4  using namespace std;
5
6  // Definition of inline function cube. Definition of function appears
7  // before function is called, so a function prototype is not required.
8  // First line of function definition also acts as the prototype.
9  inline double cube(double side) {
10    return side * side * side; // calculate cube
11 }
12
13 int main() {
14    double sideValue; // stores value entered by user
15    cout << "Enter the side length of your cube: ";
16    cin >> sideValue; // read value from user
17
18    // calculate cube of sideValue and display result
19    cout << "Volume of cube with side "
20       << sideValue << " is " << cube(sideValue) << '\n';
21 }
```

```
Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875
```

Fig. 5.7 `inline` function that calculates the volume of a cube.

5.12 References and Reference Parameters

Perf Two ways to pass arguments to functions in many programming languages are **pass-by-value** and **pass-by-reference**. With pass-by-value, a copy of the argument's value is made and passed to the called function. Changes to the copy do not affect the original variable's value in the caller. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. So far, each argument in this book has been passed by value. One disadvantage of pass-by-value for large data items is that copying data can take considerable execution time and memory space.

Reference Parameters

This section introduces reference parameters—one of two mechanisms to perform pass-by-reference.[22,23] When a variable is passed by reference, the caller gives the called function the ability to access that variable in the caller directly and to modify the variable.

22. Chapter 7 discusses pointers, which enable an alternative form of pass-by-reference.

23. In Chapter 11, we'll also discuss another form of reference called an *rvalue* reference.

Perf Sec Pass-by-reference is good for performance reasons because it can eliminate the pass-by-value overhead of copying large amounts of data. But pass-by-reference can weaken security—the called function can corrupt the caller's data.

After this section's example, we'll show how to achieve the performance advantage of pass-by-reference while simultaneously achieving the software-engineering advantage of protecting the caller's data from corruption.

A **reference parameter** is an alias for its corresponding argument in a function call. To indicate that a function parameter is passed by reference, place an ampersand (`&`) after the parameter's type in the function prototype. Use the same convention when listing the parameter's type in the function header. For example, the parameter declaration

```
int& number
```

when reading from right to left is pronounced, "`number` is a reference to an `int`." As always, the function prototype and header must agree.

In the function call, simply mention a variable by name to pass it by reference. In the called function's body, the reference parameter (e.g., `number`) refers to the original variable in the caller, which can be modified directly by the called function.

Passing Arguments by Value and by Reference

Figure 5.8 compares pass-by-value and pass-by-reference with reference parameters. The "styles" of the arguments in the calls to function `squareByValue` and function `square-ByReference` are identical—both variables are simply mentioned by name in the function calls. The compiler checks the function prototypes and definitions to determine whether to use pass-by-value or pass-by-reference.

```
1  // fig05_08.cpp
2  // Passing arguments by value and by reference.
3  #include <iostream>
4  using namespace std;
5
6  int squareByValue(int number); // prototype (for value pass)
7  void squareByReference(int& numberRef); // prototype (for reference pass)
8
9  int main() {
10    int x{2}; // value to square using squareByValue
11    int z{4}; // value to square using squareByReference
12
13    // demonstrate squareByValue
14    cout << "x = " << x << " before squareByValue\n";
```

```
15       cout << "Value returned by squareByValue: "
16          << squareByValue(x) << '\n';
17       cout << "x = " << x << " after squareByValue\n\n";
18
19       // demonstrate squareByReference
20       cout << "z = " << z << " before squareByReference\n";
21       squareByReference(z);
22       cout << "z = " << z << " after squareByReference\n";
23    }
24
25    // squareByValue multiplies number by itself, stores the
26    // result in number and returns the new value of number
27    int squareByValue(int number) {
28       return number *= number; // caller's argument not modified
29    }
30
31    // squareByReference multiplies numberRef by itself and stores the result
32    // in the variable to which numberRef refers in function main
33    void squareByReference(int& numberRef) {
34       numberRef *= numberRef; // caller's argument modified
35    }
```

```
x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference
```

Fig. 5.8 Passing arguments by value and by reference.

References as Aliases within a Function

References can also be used as aliases for other variables within a function (although they typically are used with functions as shown in Fig. 5.8). For example, the code

```
int count{1}; // declare integer variable count
int& cRef{count}; // create cRef as an alias for count
++cRef; // increment count (using its alias cRef)
```

increments `count` by using its alias `cRef`. Reference variables must be initialized in their declarations and cannot be reassigned as aliases to other variables. In this sense, references are constant. All operations performed on the alias (i.e., the reference) are actually performed on the original variable. The alias is simply another name for the original variable. Unless it's a reference to a constant (discussed below), a reference's initializer must be an *lvalue*—something that can appear on the left side of an assignment, like a variable name. A reference may not be initialized with a constant or *rvalue* expression—that is, something that may only appear on the right side of an assignment, such as the result of a calculation.

`const` References

To specify that a reference parameter should not be allowed to modify the corresponding argument in the caller, place the `const` qualifier before the type name in the parameter's declaration. For example, consider a `displayName` function:

```
void displayName(std::string name) {
   std::cout << name << '\n';
}
```
When called, it receives a copy of its `string` argument. Since `string` objects can be large, this copy operation could degrade an application's performance. For this reason, `string` objects (and objects in general) should be passed to functions by reference.

Also, the `displayName` function does not need to modify its argument. So, following the principle of least privilege, we'd declare the parameter as

```
const std::string& name
```

Perf Reading this from right to left, the `name` parameter is a reference to a `string` that is constant. We get the performance of passing the `string` by reference. Also, `displayName` treats the argument as a constant, so `displayName` cannot modify the value in the caller.

Returning a Reference to a Local Variable Is Dangerous

Err SE When returning a reference to a local non-`static` variable, the reference refers to a variable that's discarded when the function returns. Attempting to access such a variable yields undefined behavior, often crashing the program or corrupting data.[24] References to undefined variables are called **dangling references**. This is a logic error for which compilers often issue a warning. Software-engineering teams often have policies requiring that code must compile without warnings before it can be deployed. You can enforce this for your team by using your compiler's option that causes it to treat warnings as errors.

24. C++ Core Guidelines. Accessed December 28, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-dangle.

5.13 Default Arguments

It's common for a program to invoke a function from several places with the same argument value for a particular parameter. In such cases, you can specify that such a parameter has a **default argument**—that is, a default value to be passed to that parameter. When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call, inserting the default value of that argument.

`boxVolume` Function with Default Arguments

Figure 5.9 demonstrates using default arguments to calculate a box's volume. The function prototype for `boxVolume` (line 7) specifies that all three parameters have default values of `1` by placing `= 1` to the right of each parameter.

```
 1  // fig05_09.cpp
 2  // Using default arguments.
 3  #include <iostream>
 4  using namespace std;
 5
 6  // function prototype that specifies default arguments
 7  int boxVolume(int length = 1, int width = 1, int height = 1);
 8
 9  int main() {
10     // no arguments--use default values for all dimensions
```

```
11       cout << "The default box volume is: " << boxVolume();
12
13       // specify length; default width and height
14       cout << "\n\nThe volume of a box with length 10,\n"
15          << "width 1 and height 1 is: " << boxVolume(10);
16
17       // specify length and width; default height
18       cout << "\n\nThe volume of a box with length 10,\n"
19          << "width 5 and height 1 is: " << boxVolume(10, 5);
20
21       // specify all arguments
22       cout << "\n\nThe volume of a box with length 10,\n"
23          << "width 5 and height 2 is: " << boxVolume(10, 5, 2)
24          << '\n';
25    }
26
27    // function boxVolume calculates the volume of a box
28    int boxVolume(int length, int width, int height) {
29       return length * width * height;
30    }
```

```
The default box volume is: 1

The volume of a box with length 10,
width 1 and height 1 is: 10

The volume of a box with length 10,
width 5 and height 1 is: 50

The volume of a box with length 10,
width 5 and height 2 is: 100
```

Fig. 5.9 Using default arguments.

The first call to `boxVolume` (line 11) specifies no arguments, thus using all three default values of 1. The second call (line 15) passes only a `length` argument, thus using default values of 1 for the `width` and `height` arguments. The third call (line 19) passes arguments for only `length` and `width`, thus using a default value of 1 for the `height` argument. The last call (line 23) passes arguments for `length`, `width` and `height`, thus using no default values. Any arguments passed to the function explicitly are assigned to the function's parameters from left to right. Therefore, when `boxVolume` receives one argument, the function assigns the value of that argument to its `length` parameter (i.e., the leftmost parameter in the parameter list). When `boxVolume` receives two arguments, the function assigns those arguments' values to its `length` and `width` parameters in that order. Finally, when `boxVolume` receives all three arguments, the function assigns the values of those arguments to the `length`, `width` and `height` parameters, respectively.

Notes Regarding Default Arguments

Default arguments must be the rightmost (trailing) arguments in a function's parameter list. When calling a function with two or more default arguments, if an omitted argument is not the rightmost argument, then all arguments to the right of that argument also must be omitted. Default arguments must be specified with the first occurrence of the function name—typically, in the function prototype. If the function prototype is omitted because the function definition also serves as its prototype, the default arguments should be specified in the function header. Default values can be any expression, including constants, global variables or function calls. Default arguments also can be used

with `inline` functions. Using default arguments can simplify writing function calls, but some programmers feel that explicitly specifying all arguments is clearer.

5.14 Unary Scope Resolution Operator

C++ provides the **unary scope resolution operator (::)** to access a global variable when a local variable of the same name is in scope. The unary scope resolution operator cannot be used to access a local variable of the same name in an outer block. A global variable can be accessed directly without the unary scope resolution operator if the name of the global variable is not the same as that of a local variable in scope.

Err Figure 5.10 shows the unary scope resolution operator with local and global variables of the same name (lines 6 and 9). To emphasize that the local and global versions of variable `number` are distinct, the program declares one variable `int` and the other `double`. In general, you should avoid using variables of the same name for different purposes in a program. Although this is allowed in various circumstances, it can lead to errors. Generally, global variables are discouraged. If you use one, always use the unary scope resolution operator (`::`) to refer to it (even if there is no collision with a local-variable name). This makes it clear that you're accessing a global variable. It also makes programs easier to modify by reducing the risk of name collisions with nonglobal variables and eliminates logic errors that might occur if a local variable hides the global variable.

```
1  // fig05_10.cpp
2  // Unary scope resolution operator.
3  #include <iostream>
4  using namespace std;
5
6  const int number{7}; // global variable named number
7
8  int main() {
9     const double number{10.5}; // local variable named number
10
11    // display values of local and global variables
12    cout << "Local double value of number = " << number
13      << "\nGlobal int value of number = " << ::number << '\n';
14 }
```

```
Local double value of number = 10.5
Global int value of number = 7
```

Fig. 5.10 Unary scope resolution operator.

5.15 Function Overloading

C++ enables functions of the same name to be defined, as long as they have different signatures. This is called **function overloading**. The C++ compiler selects the proper function to call by examining the number, types and order of the arguments in the call. Function overloading is used to create several functions of the same name that perform similar tasks but on data of different types. For example, many functions in the math library are overloaded for different numeric types—the C++ standard requires `float`, `double` and `long double` overloaded versions of the math library functions introduced in Section 5.3. Overloading functions that perform closely related tasks can make programs clearer.

Overloaded `square` Functions

Figure 5.11 uses overloaded `square` functions to calculate the square of an `int` (lines 7–10) and the square of a `double` (lines 13–16). Line 19 invokes the `int` version by passing the literal value `7`. C++ treats whole-number literal values as `int`s. Similarly, line 21 invokes the `double` version by passing the literal value `7.5`, which C++ treats as a `double`. In each case, the compiler chooses the proper function to call, based on the type of the argument. The output confirms that the proper function was called in each case.

```
1  // fig05_11.cpp
2  // Overloaded square functions.
3  #include <iostream>
4  using namespace std;
5
6  // function square for int values
7  int square(int x) {
8     cout << "square of integer " << x << " is ";
9     return x * x;
10 }
11
12 // function square for double values
13 double square(double y) {
14    cout << "square of double " << y << " is ";
15    return y * y;
16 }
17
18 int main() {
19    cout << square(7); // calls int version
20    cout << '\n';
21    cout << square(7.5); // calls double version
22    cout << '\n';
23 }
```

```
square of integer 7 is 49
square of double 7.5 is 56.25
```

Fig. 5.11 Overloaded `square` functions.

How the Compiler Differentiates Among Overloaded Functions

Overloaded functions are distinguished by their signatures. A signature is a combination of a function's name and its parameter types (in order). **Type-safe linkage** ensures that the proper function is called and that the types of the arguments conform to the types of the parameters. To enable type-safe linkage, the compiler internally encodes each function identifier with the types of its parameters—a process referred to as **name mangling**. These encodings vary by compiler, so everything that will be linked to create an executable for a given platform must be compiled using the same compiler for that platform. Figure 5.12 was compiled with GNU C++.[25] Rather than showing the execution output of the program as we normally would, we show the mangled function names produced in assembly language by GNU C++.[26]

25. The empty-bodied `main` function ensures that we do not get a linker error if we compile this code.
26. The command `g++ -S fig05_12.cpp` produces the assembly-language file `fig05_12.s`.

```
1   // fig05_12.cpp
2   // Name mangling to enable type-safe linkage.
3
4   // function square for int values
5   int square(int x) {
6      return x * x;
7   }
8
9   // function square for double values
10  double square(double y) {
11     return y * y;
12  }
13
14  // function that receives arguments of types
15  // int, float, char and int&
16  void nothing1(int a, float b, char c, int& d) { }
17
18  // function that receives arguments of types
19  // char, int, float& and double&
20  int nothing2(char a, int b, float& c, double& d) {
21     return 0;
22  }
23
24  int main() { }
```

```
_Z6squarei
_Z6squared
_Z8nothing1ifcRi
_Z8nothing2ciRfRd
main
```

Fig. 5.12 Name mangling to enable type-safe linkage.

For GNU C++, each mangled name (other than `main`) begins with an underscore (_) followed by the letter `Z`, a number and the function name. The number that follows `Z` specifies how many characters are in the function's name. For example, function `square` has 6 characters in its name, so its mangled name is prefixed with `_Z6`. Following the function name is an encoding of its parameter list:

- For function `square` that receives an `int` (line 5), `i` represents `int`, as shown in the output's first line.
- For function `square` that receives a `double` (line 10), `d` represents `double`, as shown in the output's second line.
- For function `nothing1` (line 16), `i` represents an `int`, `f` represents a `float`, `c` represents a `char` and `Ri` represents an `int&` (i.e., a reference to an `int`), as shown in the output's third line.
- For function `nothing2` (line 20), `c` represents a `char`, `i` represents an `int`, `Rf` represents a `float&` and `Rd` represents a `double&`.

The compiler distinguishes the two `square` functions by their parameter lists—one specifies `i` for `int` and the other `d` for `double`. The return types of the functions are not specified in the mangled names. Overloaded functions can have different return types, but if they do, they must also have different parameter lists. Function-name mangling is compiler-specific. For example, Visual C++ produces the name `square@@YAHH@Z` for the square function at line 5. The GNU C++ compiler did not mangle `main`'s name, but some compilers do. For example, Visual C++ uses `_main`.

Creating overloaded functions with identical parameter lists and different return types is a compilation error. The compiler uses only the parameter lists to distinguish between overloaded functions. Such functions need not have the same number of parameters.

Err A function with default arguments omitted might be called identically to another overloaded function; this is a compilation error. For example, having a program that contains both a function that explicitly takes no arguments and a function of the same name that contains all default arguments results in a compilation error when an attempt is made to use that function name in a call passing no arguments. The compiler cannot unambiguously determine which version of the function to choose.

Overload Resolution Details

For the complete details of how compilers resolve overloaded function calls, see the C++ standard's "Overload Resolution" section at

```
https://timsong-cpp.github.io/cppwp/n4861/over.match
```

and `cppreference.com`'s "Overload resolution" section at

```
https://en.cppreference.com/w/cpp/language/overload_resolution
```

Overloaded Operators

In Chapter 11, we discuss how to overload operators to define how they should operate on objects of user-defined data types. (In fact, we've been using many overloaded operators to this point, including the stream insertion `<<` and the stream extraction `>>` operators. These are overloaded for all the fundamental types. We say more about overloading `<<` and `>>` to be able to handle objects of user-defined types in Chapter 11.)

5.16 Function Templates

20 Overloaded functions typically perform similar operations on different data types. If the program logic and operations are identical for each data type, overloading may be performed more compactly and conveniently with **function templates**. You write a single function template definition. Given the argument types provided in calls to this function, C++ automatically generates separate **function template specializations** (also called **template instantiations**) to handle each type of call appropriately. Thus, defining a single function template essentially defines a whole family of overloaded functions. Programming with templates is also known as **generic programming**. In this section, we define a custom function template. In subsequent chapters, you'll use many preexisting C++ standard library templates. In Chapter 15, we'll discuss defining custom templates in detail, including C++20 abbreviated function templates and C++20 Concepts.

`maximum` Function Template

Figure 5.13 defines a `maximum` function template that determines the largest of three values. All function template definitions begin with the `template` **keyword** (line 3) followed by a **template**

parameter list enclosed in angle brackets (`<` and `>`). Every parameter in the template parameter list is preceded by keyword `typename` or keyword `class` (they are synonyms in this context). The **type parameters** are placeholders for fundamental types or user-defined types. These placeholders—in this case, `T`—are used to specify the types of the function's parameters (line 4), to specify the function's return type (line 4) and to declare variables within the body of the function definition (line 5). A function template is defined like any other function but uses the type parameters as placeholders for actual data types.

```
1  // Fig. 5.13: maximum.h
2  // Function template maximum header.
3  template <typename T> // or template <class T>
4  T maximum(T value1, T value2, T value3) {
5     T maximumValue{value1}; // assume value1 is maximum
6
7     // determine whether value2 is greater than maximumValue
8     if (value2 > maximumValue) {
9        maximumValue = value2;
10    }
11
12    // determine whether value3 is greater than maximumValue
13    if (value3 > maximumValue) {
14       maximumValue = value3;
15    }
16
17    return maximumValue;
18 }
```

Fig. 5.13 Function template `maximum` header.

This function template `maximum`'s one type parameter `T` (line 3) is a placeholder for the type of data the function processes. Type parameter names in a particular template parameter list must be unique. When the compiler encounters a call to `maximum` in the program source code, the compiler substitutes the argument types in the `maximum` call for `T` throughout the template definition, creating a complete function template specialization that determines the maximum of three values of the specified type. The values must have the same type because we use only one type parameter in this example. Then the newly created function is compiled—templates are a means of code generation. We'll use C++ standard library templates that require multiple type parameters in Chapter 13.

Using Function Template `maximum`

Figure 5.14 uses the `maximum` function template to determine the largest of three `int` values, three `double` values and three `char` values, respectively (lines 15, 24 and 33). Each call uses arguments of a different type, so "behind the scenes" the compiler creates a separate function definition for each—one expecting three `int` values, one expecting three `double` values and one expecting three `char` values, respectively.

```
1  // fig05_14.cpp
2  // Function template maximum test program.
3  #include <iostream>
4  #include "maximum.h" // include definition of function template maximum
5  using namespace std;
6
```

```
7   int main() {
8      // demonstrate maximum with int values
9      cout << "Input three integer values: ";
10     int int1, int2, int3;
11     cin >> int1 >> int2 >> int3;
12
13     // invoke int version of maximum
14     cout << "The maximum integer value is: "
15        << maximum(int1, int2, int3);
16
17     // demonstrate maximum with double values
18     cout << "\n\nInput three double values: ";
19     double double1, double2, double3;
20     cin >> double1 >> double2 >> double3;
21
22     // invoke double version of maximum
23     cout << "The maximum double value is: "
24        << maximum(double1, double2, double3);
25
26     // demonstrate maximum with char values
27     cout << "\n\nInput three characters: ";
28     char char1, char2, char3;
29     cin >> char1 >> char2 >> char3;
30
31     // invoke char version of maximum
32     cout << "The maximum character value is: "
33        << maximum(char1, char2, char3) << '\n';
34  }
```

```
Input three integer values: 1 2 3
The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1
The maximum double value is: 3.3

Input three characters: A C B
The maximum character value is: C
```

Fig. 5.14 Function template maximum test program.

`maximum` Function Template Specialization for Type `int`

The function template specialization created for type `int` effectively replaces each occurrence of `T` with `int` as follows:

```
int maximum(int value1, int value2, int value3) {
int maximumValue{value1}; // assume value1 is maximum
   // determine whether value2 is greater than maximumValue
   if (value2 > maximumValue) {
      maximumValue = value2;
   }
   // determine whether value3 is greater than maximumValue
   if (value3 > maximumValue) {
      maximumValue = value3;
    }
    return maximumValue;
```

```
}
```

5.17 Recursion

For some problems, it's useful to have functions call themselves. A **recursive function** is a function that calls itself, either directly or indirectly (through another function). This section and the next present simple examples of recursion. Recursion is discussed at length in upper-level computer-science courses.

Recursion Concepts

We first consider recursion conceptually, then examine programs containing recursive functions. Recursive problem-solving approaches have several elements in common. A recursive function is called to solve a problem but knows how to solve only the simplest case(s), or so-called **base case(s)**. If the function is called with a base case, the function simply returns a result. If the function is called with a more complex problem, it typically divides the problem into two conceptual pieces—a piece that the function knows how to do and a piece that it does not know how to do. To make recursion feasible, the latter piece must resemble the original problem, but be a slightly simpler or smaller version. This new problem looks like the original, so the function calls a copy of itself to work on the smaller problem. This is referred to as a **recursive call** and is called the **recursion step**. The recursion step often includes the keyword `return` because its result will be combined with the portion of the problem the function knew how to solve to form the result passed back to the original caller, possibly `main`.

Err Omitting the base case or writing the recursion step incorrectly so that it does not converge on the base case causes an infinite recursion error, which typically leads to a stack overflow. This is analogous to an infinite loop in an iterative (nonrecursive) solution. Infinite recursion also occurs if a function that was not intended to be recursive accidentally calls itself, either directly or indirectly through another function.

The recursion step executes while the original call to the function is still "open," meaning it has not yet finished executing. The recursion step can result in many more such recursive calls, as the function keeps dividing each new subproblem with which the function is called into two conceptual pieces. For the recursion to eventually terminate, each time the function calls itself with a slightly simpler version of the original problem, this sequence of smaller and smaller problems must eventually converge on the base case. At that point, the function recognizes the base case and returns a result to the previous copy of the function. Then, a sequence of returns ensues until the original call eventually returns the final result to its caller.[27] This sounds quite exotic compared to the kind of problem-solving we've been using to this point. As an example of these concepts at work, let's write a recursive program to perform a popular mathematical calculation.

27. The C++ standard document indicates that `main` should not be called within a program or recursively. Its sole purpose is to be the starting point for program execution. C++ Standard. Accessed December 28, 2021. https://timsong-cpp.github.io/cppwp/n4861/basic.start.main and https://timsong-cpp.github.io/cppwp/n4861/expr.call.

Factorial

The factorial of a non-negative integer *n*, written *n*! (pronounced "*n* factorial"), is the product

$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$
with 1! equal to 1, and 0! defined to be 1. For example, 5! is the product $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120.

Iterative Factorial

The factorial of an integer, `number`, greater than or equal to 0, can be calculated **iteratively** (nonrecursively) using a `for` statement as follows:

```
int factorial{1};
for (int counter{number}; counter >= 1; --counter) {
   factorial *= counter;
}
```

Recursive Factorial

A recursive factorial definition is arrived at by observing the following algebraic relationship:

$n! = n \cdot (n-1)!$

For example, 5! is clearly equal to 5 * 4! as is shown by the following:

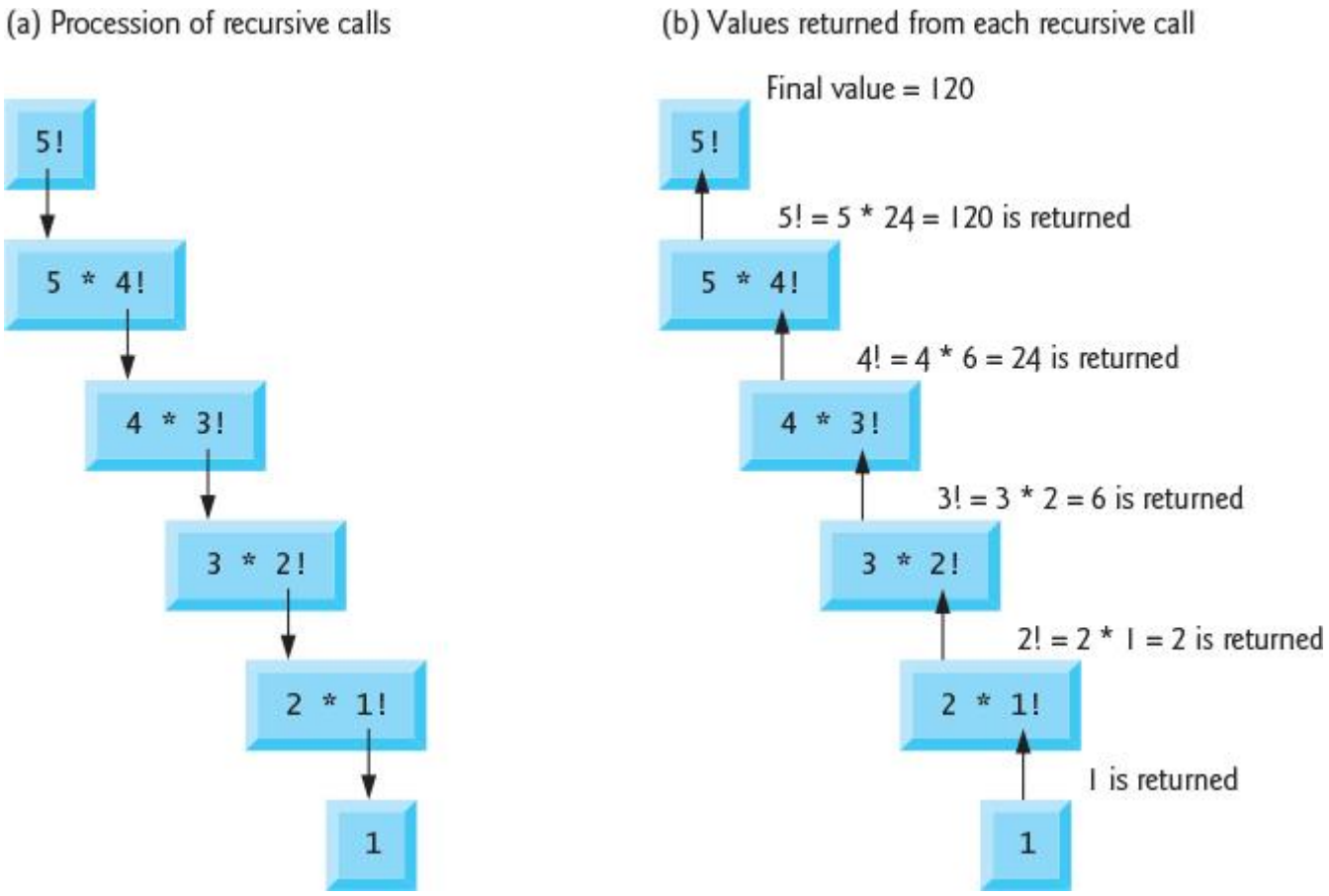
$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$

$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$

$5! = 5 \cdot (4!)$

Evaluating 5!

The evaluation of 5! would proceed as shown in the following diagram. Part (a) illustrates how the succession of recursive calls proceeds until 1! is evaluated to be 1, terminating the recursion. Part (b) of the diagram shows the values returned from each recursive call to its caller until the final value is calculated and returned.


(a) Procession of recursive calls
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
(b) Values returned from each recursive call
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 is returned
1

Using a Recursive `factorial` Function to Calculate Factorials

Figure 5.15 uses recursion to calculate and print the factorials of the integers 0–10. The recursive function `factorial` (lines 18–25) first determines whether the terminating condition `number <= 1` (i.e., the base case; line 19) is true. If `number` is less than or equal to 1, the `factorial` function returns 1 (line 20), no further recursion is necessary and the function terminates. If `number` is greater than 1, line 23 expresses the problem as the product of `number` and a recursive call to `factorial` evaluating the factorial of `number - 1`, which is a slightly simpler problem than the original calculation `factorial(number)`.

```
1  // fig05_15.cpp
2  // Recursive function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  long factorial(int number); // function prototype
8
9  int main() {
10   // calculate the factorials of 0 through 10
11   for (int counter{0}; counter <= 10; ++counter) {
12      cout << setw(2) << counter << "! = " << factorial(counter)
13         << '\n';
14   }
15 }
16
17 // recursive definition of function factorial
```

```
18   long factorial(int number) {
19      if (number <= 1) { // test for base case
20         return 1; // base cases: 0! = 1 and 1! = 1
21      }
22      else { // recursion step
23         return number * factorial(number - 1);
24      }
25   }
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 5.15 Recursive function `factorial`.

Factorial Values Grow Quickly

Function `factorial` receives a parameter of type `int` and returns a result of type `long`. Typically, a `long` is stored in at least four bytes (32 bits); such a variable can hold a value in the range –2,147,483,648 to 2,147,483,647. Unfortunately, the function `factorial` produces large values so quickly that type `long` does not help us compute many factorial values before reaching the maximum value of a `long`. For larger integer values, we could use type `long long` (Section 3.11) or a class that represents arbitrary-sized integers (such as the open-source `BigNumber` class we introduced in Section 3.12.

5.18 Example Using Recursion: Fibonacci Series

The Fibonacci series

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

The series occurs in nature and describes a form of a spiral. The ratio of successive Fibonacci numbers converges on a constant value of 1.618..., which is represented by the constant `numbers::phi` in header `<numbers>`. This number frequently occurs in nature and has been called the **golden ratio** or the **golden mean**. Humans tend to find the golden mean aesthetically pleasing. Architects often design windows, rooms and buildings whose length and width are in the ratio of the golden mean. Postcards are often designed with a golden mean length/width ratio. A web search for "Fibonacci in nature" reveals many interesting examples, including flower petals, shells, spiral galaxies, hurricanes and more.

Recursive Fibonacci Definition

The Fibonacci series can be defined recursively as follows:

fibonacci(0) = 0

fibonacci(1) = 1

fibonacci(*n*) = fibonacci(*n* – 1) + fibonacci(*n* – 2)

The program of Fig. 5.16 calculates the *n*th Fibonacci number recursively by using function `fibonacci`. Fibonacci numbers tend to become large quickly, although much more slowly than factorials. Figure 5.16 shows the execution of the program, which displays the Fibonacci values for several numbers.

```
1   // fig05_16.cpp
2   // Recursive function fibonacci.
3   #include <iostream>
4   using namespace std;
5
6   long fibonacci(long number); // function prototype
7
8   int main() {
9      // calculate the fibonacci values of 0 through 10
10     for (int counter{0}; counter <= 10; ++counter)
11        cout << "fibonacci(" << counter << ") = "
12           << fibonacci(counter) << '\n';
13
14     // display higher fibonacci values
15     cout << "\nfibonacci(20) = " << fibonacci(20) << '\n';
16     cout << "fibonacci(30) = " << fibonacci(30) << '\n';
17     cout << "fibonacci(35) = " << fibonacci(35) << '\n';
18  }
19
20  // recursive function fibonacci
21  long fibonacci(long number) {
22     if ((0 == number) || (1 == number)) { // base cases
23        return number;
24     }
25     else { // recursion step
26        return fibonacci(number - 1) + fibonacci(number - 2);
27     }
28  }
```

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
fibonacci(10) = 55

fibonacci(20) = 6765
fibonacci(30) = 832040
```

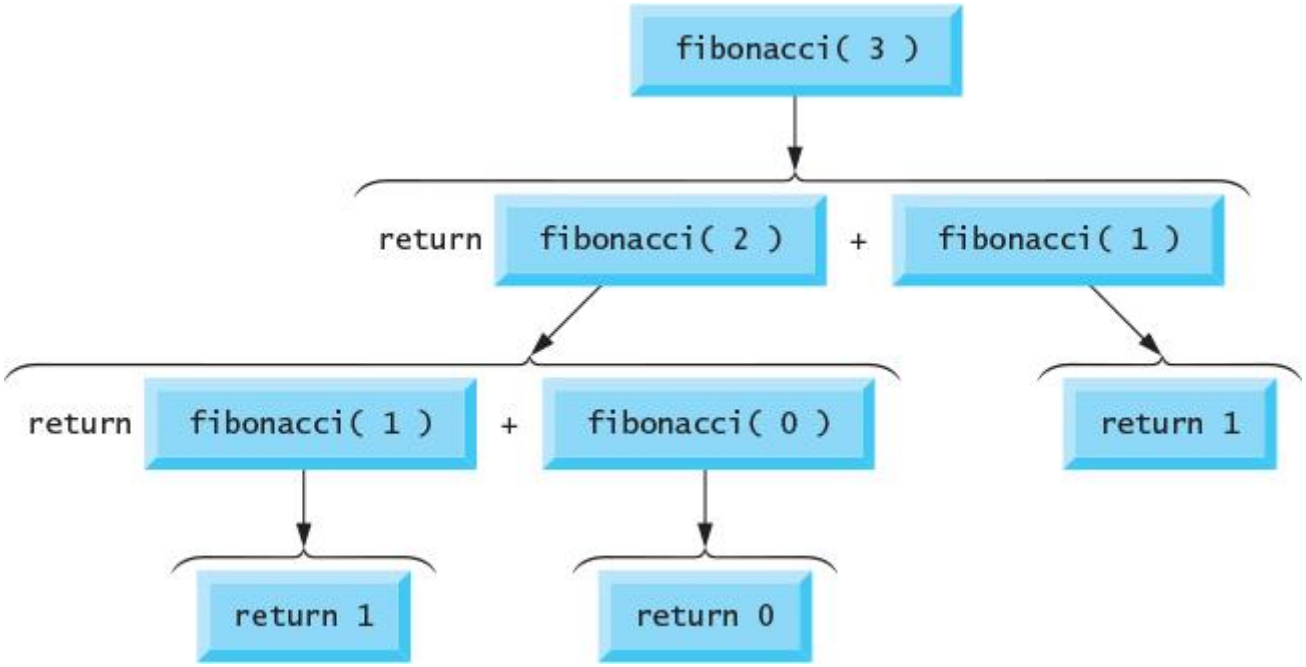
```
fibonacci(35) = 9227465
```

Fig. 5.16 Recursive function `fibonacci`.

The application begins with a loop that calculates and displays the Fibonacci values for the integers 0–10 and is followed by three calls to calculate the Fibonacci values of the integers 20, 30 and 35 (lines 15–17). The calls to `fibonacci` in `main` (lines 12 and 15–17) are not recursive calls, but the calls from line 26 of `fibonacci` are recursive. Each time the program invokes `fibonacci` (lines 21–28), the function immediately tests the base case to determine whether `number` is equal to `0` or `1` (line 22). If this is true, line 23 returns `number`. Interestingly, if `number` is greater than 1, the recursion step (line 26) generates two recursive calls, each for a slightly smaller problem than the original call to `fibonacci`.

Evaluating `fibonacci(3)`

The following diagram shows how function `fibonacci` would evaluate `fibonacci(3)`. This figure raises some interesting issues about the order in which C++ compilers evaluate the operands of operators. This is a separate issue from the order in which operators are applied to their operands, which is dictated by the rules of operator precedence and grouping. The following diagram shows that evaluating `fibonacci(3)` causes two recursive calls, namely, `fibonacci(2)` and `fibonacci(1)`. In what order are these calls made?



Order of Evaluation of Operands

Most programmers simply assume that the operands are evaluated left-to-right, which is the case in some programming languages. C++ does not specify the order in which the operands of many operators (including `+`) are to be evaluated. Therefore, you must make no assumption about the order in which these calls execute. The calls could, in fact, execute `fibonacci(2)` first, then `fibonacci(1)`, or `fibonacci(1)` first, then `fibonacci(2)`. In this program and in most others, it turns out that the final result would be the same. However, in some programs, the evaluation of an operand can have side effects (input/output operations or changes to data values) that could affect the expression's final result.

Operators for Which Order Of Evaluation Is Specified

Before C++17, C++ specified the order of evaluation of the operands of only the operators `&&`, `||`, comma (`,`) and `?:`. The first three are binary operators whose two operands are guaranteed to be

evaluated left-to-right. The last operator is C++'s only ternary operator. Its leftmost operand is always evaluated first. If it evaluates to true, the middle operand evaluates next, and the last operand is ignored. If the leftmost operand evaluates to false, the third operand evaluates next, and the middle operand is ignored.

17 C++17 also specifies the order of evaluation of the operands for various other operators. For the operators `[]` (Chapter 6), `->` (Chapter 7), parentheses (of a function call), `<<`, `>>`, `.*`, and `->*`, the compiler evaluates the operands left-to-right. For a function call's parentheses, this means that the compiler evaluates the function name before the arguments. The compiler evaluates the operands of assignment operators right-to-left.

Writing programs that depend on the order of evaluation of the operands of operators can lead to logic errors. To ensure that side effects are applied in the correct order, break complex expressions into separate statements. Recall that the `&&` and `||` operators use short-circuit evaluation. Placing an expression with a side effect on the right side of the `&&` or `||` operator is a logic error if that expression should always be evaluated.

Exponential Complexity

A word of caution is in order about recursive programs like the one we use here to generate Fibonacci numbers. Each level of recursion in function `fibonacci` has a doubling effect on the number of function calls; that is, the number of recursive calls that are required to calculate the *n*th Fibonacci number is on the order of 2^n. This rapidly gets out of hand. Calculating only the 20th Fibonacci number would require on the order of 2^{20} or about a million calls, calculating the 30th Fibonacci number would require on the order of 2^{30} or about a billion calls, and so on. Computer scientists refer to this as **exponential complexity**. Problems of this nature can humble even the world's most powerful computers as *n* becomes large. Complexity issues in general, and exponential complexity in particular, are discussed in detail in the upper-level computer-science course typically called Algorithms. Avoid Fibonacci-style recursive programs that result in an exponential "explosion" of calls.

5.19 Recursion vs. Iteration

In the two prior sections, we studied two recursive functions that can also be implemented with simple iterative programs. This section compares the two approaches and discusses why you might choose one approach over the other in a particular situation.

- **Both iteration and recursion are based on a control statement.** Iteration uses an iteration statement. Recursion uses a selection statement.
- **Both iteration and recursion involve iteration.** Iteration explicitly uses an iteration statement. Recursion achieves iteration through functions that call themselves.
- **Iteration and recursion each have a termination test.** Iteration terminates when the loop-continuation condition fails. Recursion terminates when a base case is recognized.
- **Counter-controlled iteration and recursion each gradually approach termination.** Iteration modifies a counter until the counter assumes a value that makes the loop-continuation condition fail. Recursion produces simpler versions of the original problem until the base case is reached.
- Err **Both iteration and recursion can occur infinitely.** An infinite loop occurs with iteration if the loop-continuation test never becomes false. Infinite recursion occurs if the recursion step does not reduce the problem during each recursive call in a manner that converges on the base case.

Iterative Factorial Implementation

Let's examine the differences between iteration and recursion with an iterative version of calculating factorials (Fig. 5.17). Lines 22–24 use an iteration statement rather than the recursive solution's selection statement (Fig. 5.15, lines 19–24). In Fig. 5.15's recursive solution, line 19 tests the base case to terminate recursion. In Fig. 5.17's iterative solution, line 22 tests the loop-continuation condition—if the test fails, the loop terminates. Also, rather than producing simpler versions of the original problem, the iterative solution uses a counter that's modified until the loop-continuation condition becomes false.

```
1  // fig05_17.cpp
2  // Iterative function factorial.
3  #include <iostream>
4  #include <iomanip>
5  using namespace std;
6
7  long factorial(int number); // function prototype
8
9  int main() {
10    // calculate the factorials of 0 through 10
11    for (int counter{0}; counter <= 10; ++counter) {
12       cout << setw(2) << counter << "! = " << factorial(counter)
13          << '\n';
14    }
15 }
16
17 // iterative function factorial
18 long factorial(int number) {
19    long result{1};
20
21    // iterative factorial calculation
22    for (int i{number}; i >= 1; --i) {
23       result *= i;
24    }
25
26    return result;
27 }
```

```
 0! = 1
 1! = 1
 2! = 2
 3! = 6
 4! = 24
 5! = 120
 6! = 720
 7! = 5040
 8! = 40320
 9! = 362880
10! = 3628800
```

Fig. 5.17 Iterative function `factorial`.

Negatives of Recursion

Perf Recursion repeatedly invokes the mechanism, and consequently the overhead, of function calls. This can be expensive in both processor time and memory space. Each recursive call creates another copy of the function's variables; this can consume considerable memory. Iteration normally occurs within a function, so the overhead of repeated function calls and extra memory assignment is omitted. So why choose recursion?

When to Choose Recursion vs. Iteration

Perf Any problem that can be solved recursively can also be solved iteratively (nonrecursively). A recursive approach is normally chosen when the recursive approach more naturally mirrors the problem and results in a program that's easier to understand and debug. Another reason to choose a recursive solution is that an iterative solution may not be apparent when a recursive solution is. If possible, avoid using recursion in performance situations. Recursive calls take time and consume additional memory.

5.20 Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz

No doubt, you've noticed that the last Objectives bullet for this chapter, the last section name in the chapter outline, the last sentence in Section 5.1 and the section title above all look like gibberish. These are not mistakes! In this section, we continue our Objects-Natural presentation. You'll conveniently encrypt and decrypt messages with an object you create of a preexisting class that implements a **Vigenère secret key cipher**.[28]

28. "Vigenère Cipher," Wikipedia. Wikimedia Foundation. Accessed December 26, 2021. https://en.wikipedia.org/wiki/Vigenère_cipher.

In prior Objects-Natural sections, you created objects of built-in C++ standard library class `string` and objects of classes from open-source libraries. Sometimes you'll use classes built by your organization or team members for internal use or for use in a specific project. For this example, we wrote our own `Cipher` class (in the header `"cipher.h"`) and provided it to you. In Chapter 9, Custom Classes, you'll build your own custom classes.

Cryptography

Sec Cryptography has been in use for thousands of years[29,30] and is critically important in today's connected world. Every day, cryptography is used behind the scenes to ensure that your Internet-based communications are private and secure. For example, most websites now use the HTTPS protocol to encrypt and decrypt your web interactions.

29. "Cryptography." Wikipedia. Wikimedia Foundation, May 14, 2020. https://en.wikipedia.org/wiki/Cryptography#History_of_cryptography_and_cryptanalysis.
30. Binance Academy. "History of Cryptography." Binance Academy. Binance Academy, January 19, 2020. https://www.binance.vision/security/history-of-cryptography.

Caesar Cipher

Julius Caesar used a simple **substitution cipher** to encrypt military communications.[31] His technique—known as the **Caesar cipher**—replaces every letter in a communication with the letter three ahead in

the alphabet. So, A is replaced with D, B with E, C with F, … X with A, Y with B and Z with C. Thus, the plaintext

31. “Caesar Cipher.” Wikipedia. Wikimedia Foundation, May 7, 2020. https://en.wikipedia.org/wiki/Caesar_cipher.

```
Caesar Cipher
```

would be encrypted as

```
Fdhvdu Flskhu
```

The encrypted text is known as the **ciphertext**.

For a fun way to play with the Caesar cipher and many other cipher techniques, check out the website

```
https://cryptii.com/pipes/caesar-cipher
```

which is an online implementation of the open-source `cryptii` project:

```
https://github.com/cryptii/cryptii
```

Vigenère Cipher

Simple substitution ciphers like the Caesar cipher are relatively easy to decrypt. For example, the letter “e” is the most frequently used letter in English. So, you could study ciphertext encrypted with the Caesar cipher and assume with a high likelihood that the character appearing most frequently is probably an “e.”

In this example, you’ll use a Vigenère cipher, which is a secret-key substitution cipher. This cipher is implemented using 26 Caesar ciphers—one for each letter of the alphabet. A Vigenère cipher uses letters from the plaintext and secret key to look up replacement characters in the various Caesar ciphers. You can read more about the implementation at

```
https://en.wikipedia.org/wiki/Vigen%C3%A8re_cipher
```

For this cipher, the secret key must be composed of letters. Like passwords, the secret key should not be easy to guess. We used the 11 randomly selected characters

```
XMWUJBVYHXZ
```

There's no limit to the number of characters you can use in your secret key. However, the person decrypting the ciphertext must know the secret key that was originally used to create the ciphertext.[32] Presumably, you'd provide that in advance—possibly in a face-to-face meeting.

32. There are many websites offering Vigenère cipher decoders that attempt to decrypt ciphertext without the original secret key. We tried several, but none restored our original text.

Using Our `Cipher` Class

For the example in Fig. 5.18, you'll use our class `Cipher`, which implements the Vigenère cipher. The header `"cipher.h"` (line 3) from this chapter's `ch05` examples folder defines the class. You don't need to read and understand the class's code to use its encryption and decryption capabilities. You simply create an object of class `Cipher` then call its `encrypt` and `decrypt` member functions to encrypt and decrypt text, respectively.

```
1  // fig15_18.cpp
2  // Encrypting and decrypting text with a Vigenère cipher.
3  #include "cipher.h"
4  #include <iostream>
5  #include <string>
6  using namespace std;
7
8  int main() {
9     string plainText;
10    cout << "Enter the text to encrypt:\n";
11    getline(cin, plainText);
12
13    string secretKey;
14    cout << "\nEnter the secret key:\n";
15    getline(cin, secretKey);
16
17    Cipher cipher;
18
19    // encrypt plainText using secretKey
20    string cipherText{cipher.encrypt(plainText, secretKey)};
21    cout << "\nEncrypted:\n " << cipherText << '\n';
22
23    // decrypt cipherText
24    cout << "\nDecrypted:\n "
25        << cipher.decrypt(cipherText, secretKey) << '\n';
26
27    // decrypt ciphertext entered by the user
28    cout << "\nEnter the ciphertext to decipher:\n";
29    getline(cin, cipherText);
30    cout << "\nDecrypted:\n "
31       << cipher.decrypt(cipherText, secretKey) << '\n';
32 }
```

```
Enter the text to encrypt:
Welcome to Modern C++ application development with C++20!

Enter the secret key:
XMWUJBVYHXZ
```

```
Encrypted:
   Tqhwxnz rv Jnaqnh L++ bknsfbxfeiw eztlinmyahc xdro Z++20!

Decrypted:
   Welcome to Modern C++ application development with C++20!

Enter the ciphertext to decipher:
Lnfylun Lhqtomh Wjtz Qarcv: Qjwazkrplm xzz Xndmwwqhlz

Decrypted:
   Objects Natural Case Study: Encryption and Decryption
```

Fig. 5.18 Encrypting and decrypting text with a Vigenère cipher.

Class `Cipher`'s Member Functions

The class provides two key member functions:

- `encrypt` receives `string`s representing the plaintext to encrypt and the secret key, uses the Vigenère cipher to encrypt the text, then returns a `string` containing the ciphertext.
- `decrypt` receives `string`s representing the ciphertext to decrypt and the secret key, reverses the Vigenère cipher to decrypt the text, then returns a `string` containing the plaintext.

The program first asks you to enter text to encrypt and a secret key. Line 17 creates the `Cipher` object. Lines 20–21 encrypt the text you entered and display the encrypted text. Then, lines 24–25 decrypt the text to show you the plaintext string you entered.

Though the last Objectives bullet in this chapter, the last sentence of Section 5.1 and this section's title look like gibberish, they're each ciphertext that we created with our `Cipher` class and the secret key

```
XMWUJBVYHXZ
```

Line 28–29 prompt for and input existing ciphertext, then lines 30–31 decrypt the ciphertext and display the original plaintext that we encrypted.

5.21 Wrap-Up

In this chapter, we presented function features, including function prototypes, function signatures, function headers and function bodies. We overviewed the math library functions and new math functions and constants added in C++20, C++17 and C++11.

You learned about argument coercion—forcing arguments to the appropriate types specified by the parameter declarations of a function. We presented an overview of the C++ standard library's headers. We demonstrated how to generate C++11 nondeterministic random numbers. We defined sets of constants with scoped `enum`s and introduced C++20's `using enum` declaration.

You learned about the scope of variables. We discussed two ways to pass arguments to functions—pass-by-value and pass-by-reference. We showed how to implement inline functions and functions that receive default arguments. You learned that overloaded functions have the same name but different signatures. Such functions can be used to perform the same or similar tasks, using different types or different numbers of parameters. We demonstrated using function templates to conveniently generate families of overloaded functions. You then studied recursion, where a function

calls itself to solve a problem. Finally, the Objects-Natural case study explored secret-key substitution ciphers for encrypting and decrypting text.

In Chapter 6, you'll learn how to maintain lists and tables of data in arrays and object-oriented `vector`s. You'll see a more elegant array-based implementation of the dice-rolling application.

6. arrays, vectors, Ranges and Functional-Style Programming

Objectives

In this chapter, you'll:

- Use C++ standard library class template array—a fixed-size collection of related, indexable data items.
- Declare arrays, initialize arrays and refer to the elements of arrays.
- Use the range-based for statement to reduce iteration errors.
- Pass arrays to functions.
- Sort array elements in ascending order.
- Quickly determine whether a sorted array contains a specific value using the high-performance binary_search function.
- Declare and manipulate multidimensional arrays.
- Use C++20's ranges with functional-style programming.
- Continue our Objects-Natural approach with a case study using the C++ standard library's class template vector—a variable-size collection of related data items.

Outline

6.1 Introduction
6.2 arrays
6.3 Declaring arrays
6.4 Initializing array Elements in a Loop
6.5 Initializing an array with an Initializer List
6.6 C++11 Range-Based for and C++20 Range-Based for with Initializer
6.7 Calculating array Element Values and an Intro to constexpr
6.8 Totaling array Elements
6.9 Using a Primitive Bar Chart to Display array Data Graphically
6.10 Using array Elements as Counters
6.11 Using arrays to Summarize Survey Results
6.12 Sorting and Searching arrays
6.13 Multidimensional arrays
6.14 Intro to Functional-Style Programming
6.14.1 What vs. How
6.14.2 Passing Functions as Arguments to Other Functions: Introducing Lambda Expressions
6.14.3 Filter, Map and Reduce: Intro to C++20's Ranges Library
6.15 Objects-Natural Case Study: C++ Standard Library Class Template vector
6.16 Wrap-Up

6.1 Introduction

This chapter introduces **data structures**—collections of related data items. The C++ standard library refers to data structures as **containers**. We discuss two containers:

- 11 fixed-size C++11 arrays[1] and

1. In Chapter 7, you'll see that C++ also provides a language element called an array (different from the `array` container). Modern C++ `code` should use C++11's `array` class template rather than traditional arrays.

- resizable `vectors` that can grow and shrink dynamically at execution time.

To use them, you must include the `<array>` and `<vector>` headers, respectively.

After discussing how `arrays` are declared, created and initialized, we demonstrate various `array` manipulations. We show how to search `arrays` to find particular items and sort `arrays` to put their data in ascending order. We show that attempting to access data that is not within an `array`'s or `vector`'s bounds might cause an exception—a runtime indication that a problem occurred. Then we use exception handling to resolve (or handle) that exception. Chapter 12 covers exceptions in more depth.

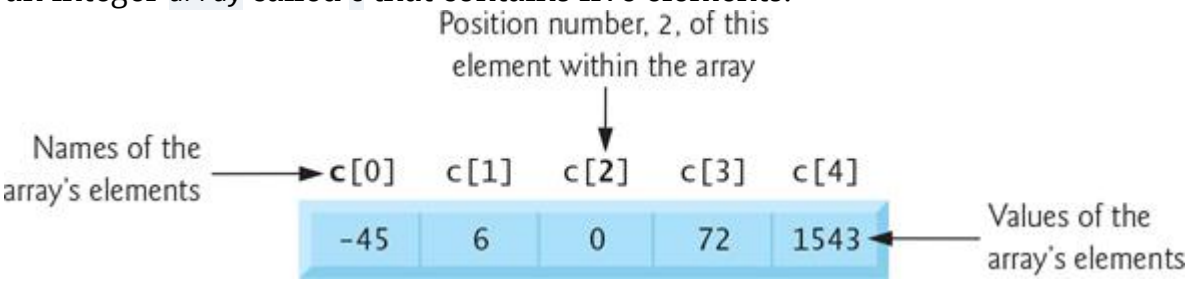
Perf Like many modern languages, C++ offers "functional-style" programming features. These can help you write more concise code that's less likely to contain errors and easier to read, debug and modify. Functional-style programs also can be easier to parallelize to improve performance on today's multi-core processors. We introduce functional-style programming with C++20's new `<ranges>` library. Finally, we continue our Objects-Natural presentation with a case study that creates and manipulates objects of the C++ standard library's class template `vector`. After reading this chapter, you'll be familiar with two array-like collections—`arrays` and `vectors`.

Fully Qualified Standard Library Names

Since Section 2.6, we've included `"using namespace std;"` in each program, so you did not need to qualify every C++ standard library feature with `"std::"`. In larger systems, `using` directives can lead to subtle, difficult-to-find bugs. Going forward, we'll fully qualify most identifiers from the C++ standard library.

6.2 `array`s

An `array`'s **elements** (data items) are arranged contiguously in memory. The following diagram shows an integer `array` called `c` that contains five elements:



One way to refer to an `array` element is to specify the `array` **name** followed by the element's **position number** in square brackets (`[]`). The position number is more formally called an **index** or **subscript**. The first element has the index `0` (zero). Thus, the elements of `array` `c` are `c[0]` (pronounced "`c` sub zero") through `c[4]`. The **value** of `c[0]` is `-45`, `c[2]` is `0` and `c[4]` is `1543`.

The highest index (in this case, 4) is always one less than the `array`'s number of elements (in this case, 5). Each `array` knows its own size, which you can get via its `size` member function, as in

```
c.size()
```

An index must be an integer expression. The brackets that enclose an index are an operator with the same precedence as a function call's parentheses. The result of that operator is an *lvalue*—it can be

used on the left side of an assignment, just as a variable name can. For example, the following statement replaces `c[4]`'s value:

```
c[4] = 87;
```

See Appendix A for the complete operator precedence chart.

6.3 Declaring `array`s

To specify an `array`'s element type and number of elements, use a declaration of the form

```
std::array<type, arraySize> arrayName;
```

The notation `<`*type*`,` *arraySize*`>` indicates that `array` is a **class template**. Like function templates, the compiler can use class templates to create **class template specializations** for various types—such as an `array` of five `int`s, an `array` of seven `double`s or an `array` of 100 `Employee`s. You'll begin creating custom types in Chapter 9. The compiler reserves the appropriate amount of memory, based on the *type* and *arraySize*. To tell the compiler to reserve five elements for integer array `c`, use this declaration:

```
std::array<int, 5> c; // c is an array of 5 int values
```

6.4 Initializing `array` Elements in a Loop

The program in Fig. 6.1 declares five-element integer `array values` (line 8). Line 5 includes the `<array>` header, which contains the definition of class template `array`.

```
1  // fig06_01.cpp
2  // Initializing an array's elements to zeros and printing the array.
3  #include <fmt/format.h> // C++20: This will be #include <format>
4  #include <iostream>
5  #include <array>
6
7  int main() {
8     std::array<int, 5> values; // values is an array of 5 int values
9
10    // initialize elements of array values to 0
11    for (size_t i{0}; i < values.size(); ++i) {
12       values[i] = 0; // set element at location i to 0
13    }
14
15    std::cout << fmt::format("{:>7}{:>10}\n", "Element", "Value");
16
17    // output each array element's value
18    for (size_t i{0}; i < values.size(); ++i) {
19       std::cout << fmt::format("{:>7}{:>10}\n", i, values[i]);
20    }
21
22    std::cout << fmt::format("\n{:>7}{:>10}\n", "Element", "Value");
23
24    // access elements via the at member function
25    for (size_t i{0}; i < values.size(); ++i) {
```

```
26          std::cout << fmt::format("{:>7}{:>10}\n", i, values.at(i));
27       }
28
29       // accessing an element outside the array's bounds with at
30       values.at(10); // throws an exception
31    }
```

```
Element  Value
      0      0
      1      0
      2      0
      3      0
      4      0

Element  Value
      0      0
      1      0
      2      0
      3      0
      4      0
terminate called after throwing an instance of
  what():  array::at: __n (which is 10) >= _Nm (which is 5)
Aborted
```

Fig. 6.1 Initializing an `array`'s elements to zeros and printing the `array`.

Assigning Values to `array` Elements in a Loop

Sec Lines 11–13 use a `for` statement to assign `0` to each `array` element. Like other non-`static` local variables, `array` elements are not implicitly initialized to zero (but elements of `static` `array`s are). In a loop that processes `array` elements, ensure that the loop-termination condition prevents accessing elements outside the array's bounds. Section 6.6 presents the range-based `for` statement, which provides a safer way to process every element of an `array`.

Type size_t

Lines 11, 18 and 25 declare each loop's control variable as type `size_t`. The C++ standard specifies that `size_t` is an unsigned integral type that can represent the size of any object.[2] Use this type for any variable representing an `array`'s size or indices. Type `size_t` is defined in the `std` namespace and is in header `<cstddef>`, which typically is included for you by other standard library headers that use `size_t`. When you compile a program, if you receive errors indicating `size_t` is not defined, simply add `#include <cstddef>` to your program.

2. C++ Standard, "17.2.4 Sizes, Alignments, and Offsets." Accessed December 29, 2021. https://timsong-cpp.github.io/cppwp/n4861/support.types#layout-3.

Displaying the `array` Elements

20 The first output statement (line 15) displays the column headings for the rows printed in the subsequent `for` statement (lines 18–20). These output statements use the C++20 text-formatting features introduced in Section 4.14 to output the `array` in tabular format.

Avoiding a Security Vulnerability: Bounds Checking for `array` Indicies

Sec When you use the `[]` operator to access an array element (as in lines 11–13 and 18–20), C++ provides no automatic array **bounds checking** to prevent you from referring to an element that does

not exist. Thus, an executing program can "walk off" either end of an `array` without warning. Class template `array`'s `at` **member function**, however, performs bounds checking. Lines 25–27 demonstrate accessing elements' values via the `at` member function. You also can assign to `array` elements by using `at` on the left side of an assignment, as in

```
values.at(0) = 10;
```

Err Line 30 attempts to access an element outside the `array`'s bounds. When the member function `at` encounters an out-of-bounds index, it generates a runtime error known as an **exception**. In this program, which we compiled with GNU g++ and ran on Linux, line 30 resulted in the following runtime error message then the program terminated:

```
terminate called after throwing an instance of
  what():  array::at: __n (which is 10) >= _Nm (which is 5)
Aborted
```

This error message indicates that the `at` member function (`array::at`) checks whether a variable named `__n` (which is `10`) is greater than or equal to a variable named `_Nm` (which is `5`). In this case, the index is out of bounds. In GNU's implementation of `array`'s `at` member function, `__n` represents the element's index, and `_Nm` represents the `array`'s size. Section 6.15 introduces how to use exception handling to deal with runtime errors. Chapter 12 covers exception handling in depth.

Sec Allowing programs to read from or write to `array` elements outside the bounds of `arrays` are common security flaws. Reading from out-of-bounds `array` elements can cause a program to crash or even appear to execute correctly while using bad data. Writing to an out-of-bounds element (known as a buffer overflow[3]) can corrupt a program's data in memory and crash a program. In some cases, attackers can exploit buffer overflows by overwriting the program's executable code with malicious code.

3. For more information on buffer overflows, see http://en.wikipedia.org/wiki/Buffer_overflow.

6.5 Initializing an `array` with an Initializer List

The elements of an `array` also can be initialized in the `array` declaration by following the `array` name with a brace-delimited comma-separated list of **initializers**. The program in Fig. 6.2 uses **initializer lists** to initialize an `array` of `int`s with five values (line 8) and an `array` of `double`s with four values (line 18) and displays each `array`'s contents:

- Line 8 explicitly specifies the `array`'s element type (`int`) and size (`5`).
- 17 Line 18 lets the compiler infer (i.e., determine) the `array`'s element type (`double`) from the values in the initializer list and the `array`'s size (`4`) from the number of initializers. This C++17 capability is known as **class template argument deduction (CTAD)**.[4] When you know the container's initial element values in advance, CTAD simplifies your code.

4. "Class Template Argument Deduction (CTAD)." Accessed December 30, 2021. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

```
1  // fig06_02.cpp
2  // Initializing an array in a declaration.
3  #include <fmt/format.h>
```

```
#include <iostream>
#include <array>

int main() {
   std::array<int, 5> values{32, 27, 64, 18, 95}; // braced initializer

   // output each array element's value
   for (size_t i{0}; i < values.size(); ++i) {
      std::cout << fmt::format("{}  ", values.at(i));
   }

   std::cout << "\n\n";

   // using class template argument deduction to determine values2's type
   std::array values2{1.1, 2.2, 3.3, 4.4};

   // output each array element's value
   for (size_t i{0}; i < values.size(); ++i) {
      std::cout << fmt::format("{}  ", values.at(i));
   }

   std::cout << "\n";
}
```

```
32 27 64 18 95

1.1 2.2 3.3 4.4
```

Fig. 6.2 Initializing an `array` in a declaration.

Fewer Initializers Than `array` Elements

If there are fewer initializers than `array` elements, the remaining `array` elements are **value initialized**—fundamental numeric types are set to `0`, `bool`s are set to `false`, and as we'll see in Chapter 9, objects receive the default initialization specified by their class definitions. For example, the following initializes an `int array`'s elements to zero

```
std::array<int, 5> values{}; // initialize elements to 0
```

because there are fewer initializers (none in this case) than `array` elements. This technique also can be used to "reinitialize" an existing `array`'s elements at execution time, as in

```
values = {}; // set all elements of values to 0
```

More Initializers Than `array` Elements

When you explicitly state the `array`'s size and use an initializer list, the number of initializers must be less than or equal to the size. The `array` declaration

```
std::array<int, 5> values{32, 27, 64, 18, 95, 14};
```

Err causes a compilation error because there are six initializers and only five `array` elements.

11 SE It's common to process all the elements of an `array`. The C++11 **range-based `for` statement** allows you to do this without using a counter. In general, when processing all elements of an `array`, use the range-based `for` statement because it ensures that your code does not "step outside" the `array`'s bounds. At the end of this section, we'll compare the **counter-controlled `for`** and **range-based `for`** statements. Figure 6.3 uses the range-based `for` to display an `array`'s contents (lines 12–14 and 23–25) and to multiply each of the `array`'s element values by 2 (lines 17–19).

```
1  // fig06_03.cpp
2  // Using range-based for.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <array>
6
7  int main() {
8     std::array items{1, 2, 3, 4, 5}; // type inferred as array<int, 5>
9
10    // display items before modification
11    std::cout << "items before modification: ";
12    for (const int& item : items) { // item is a reference to a const int
13       std::cout << fmt::format("{} ", item);
14    }
15
16    // multiply the elements of items by 2
17    for (int& item : items) { // item is a reference to an int
18       item *= 2;
19    }
20
21    // display items after modification
22    std::cout << "\nitems after modification: ";
23    for (const int& item : items) {
24       std::cout << fmt::format("{} ", item);
25    }
26
27    // sum elements of items using range-based for with initialization
28    std::cout << "\n\ncalculating a running total of items' values:\n";
29    for (int runningTotal{0}; const int& item : items) {
30       runningTotal += item;
31       std::cout << fmt::format("item: {}; running total: {}\n",
32          item, runningTotal);
33    }
34 }
```

```
items before modification: 1 2 3 4 5
items after modification: 2 4 6 8 10

calculating a running total of items
item: 2; running total: 2
item: 4; running total: 6
item: 6; running total: 12
item: 8; running total: 20
item: 10; running total: 30
```

Fig. 6.3 Using range-based `for`.

Using the Range-Based `for` to Display an `array`'s Contents

The range-based `for` statement simplifies the code for iterating through an `array`. Line 12 can be read as "for each `item` in `items`," perform some work. In each iteration, the loop assigns the next `items` element to the `const int` reference `item`, then executes the loop's body. In each iteration, `item` refers to one element value (but not an index) in `items`. In the range-based `for`'s header, you declare a **range variable** to the left of the colon (`:`) and specify an array's name to the right.[5] Here we declare the range variable `item` as a `const` reference. Recall that a reference is an alias for another variable in memory—in this case, one of the `array`'s elements.

5. You can use the range-based `for` statement with most of the C++ standard library's containers, which we discuss in Chapter 13.

Perf Declaring a range variable as a `const` reference can improve performance. A reference prevents the loop from copying each value into the range variable. This is particularly important when manipulating large objects.

Using the Range-Based `for` to Modify an `array`'s Contents

Lines 17–19 use a range-based `for` statement to multiply each element of `items` by 2. In line 17, the range variable's declaration indicates that `item` is an `int&`—that is, a reference to an `int`. This is not a `const` reference, so any change you make to `item` changes the value of the corresponding `array` element.

20 C++20 Range-Based `for` with Initializer

C++20 adds the **range-based `for` with initializer** statement. Like the `if` and `switch` statements with initializers, variables defined in a range-based `for`'s initializer exist only until the loop terminates. The initializer in line 29 defines `runningTotal` and sets it to `0`, then lines 29–33 calculate the running total of `items`' elements. The variable `runningTotal` goes out of scope at the end of the loop—that is, it no longer exists.

Avoiding Indices

In most cases, a range-based `for` statement can be used instead of the counter-controlled `for` statement. For example, totaling the integers in an `array` requires access only to the element values—their index positions in the `array` are irrelevant.

External vs. Internal Iteration

In this program, lines 12–14 are functionally equivalent to the following counter-controlled iteration:

```
for (int counter{0}; counter < items.size(); ++counter) {
  std::cout << fmt::format("{} ", items[counter]);
}
```

This style of iteration is known as **external iteration** and is error-prone. As implemented, this loop requires a control variable (`counter`) that the code *mutates* (modifies) during each loop iteration. Every time you write code that modifies a variable, it's possible to introduce an error into your code. There are several opportunities for error in the preceding code. For example, you could:

- initialize the `for` loop's control variable `counter` incorrectly,
- use the wrong loop-continuation condition or
- increment the control variable `counter` incorrectly.

Each of these could result in accessing elements outside the `array` items' bounds.

The range-based `for` statement, on the other hand, uses **internal iteration**. It hides the iteration details from you. You specify what `array` the range-based `for` should process. It knows how to get each value from the `array` and stop iterating when there are no more values.

6.7 Calculating `array` Element Values and an Intro to constexpr

Figure 6.4 sets the elements of a 5-element `array` named `values` to the even integers `2`, `4`, `6`, `8` and `10` (lines 13–15) and prints the `array` (lines 18–20). Line 14 generates values by multiplying each successive value of the loop counter, `i`, by `2` and adding `2`.

```
1  // fig06_04.cpp
2  // Set array values to the even integers from 2 to 10.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <array>
6
7  int main() {
8     // constant can be used to specify array size
9     constexpr size_t arraySize{5}; // must initialize in declaration
10
11    std::array<int, arraySize> values{}; // array values has 5 elements
12
13    for (size_t i{0}; i < values.size(); ++i) { // set the values
14       values.at(i) = 2 + 2 * i;
15    }
16
17    // output contents of array values in tabular format
18    for (const int& value : values) {
19       std::cout << fmt::format("{} ", value);
20    }
21
22    std::cout << '\n';
23 }
2 4 6 8 10
```

Fig. 6.4 Set `array values` to the even integers from 2 to 10.

Constants

11 Perf In Chapter 5, we introduced the `const` qualifier to indicate that a variable's value does not change after it is initialized. C++11 introduced the **`constexpr` qualifier** to declare variables that can be evaluated at compile-time and result in a constant. This allows the compiler to perform additional optimizations, improving application performance because there's no runtime overhead.
A `constexpr` variable is implicitly `const`. The primary difference between them is that a `constexpr` variable must be initialized at compile-time, but a `const` variable can be initialized at execution time.

Line 9 uses `constexpr` to declare the constant `arraySize` with the value `5`. A variable declared `constexpr` or `const` must be initialized when it's declared; otherwise, a compilation error occurs. Attempting to modify `arraySize` after it's initialized, as in

```
arraySize = 7;
```

results in a compilation error.[6]

6. In error messages, some compilers refer to a `const` fundamental-type variable as a "`const` object." The C++ standard defines an "object" as any "region of storage." Like class objects, fundamental-type variables also occupy space in memory, so they're often referred to as "objects" as well.

Defining the size of an array as a constant instead of a literal value makes programs clearer and easier to update. This technique eliminates **magic numbers**—numeric literal values that do not provide you with any context that helps you understand their meaning.

Using a constant allows you to provide a name for a literal value and can help explain that value's purpose in the program.

constexpr and Compile-Time

Perf 20 17 14 A modern C++ theme is to do more at compile-time for better type-checking and better runtime performance. Each new C++ standard since C++11 has expanded `constexpr`'s use. For example, in later chapters, you'll see that `constexpr` can be applied to functions. In a `constexpr` function call, if the arguments also are `constexpr`, the compiler can evaluate the call at compile-time to produce a constant. This eliminates the overhead of runtime function calls, thus further improving application performance. In C++20, C++17 and C++14, many functions throughout the C++ standard library have been declared `constexpr`.

For more details on the various uses of `constexpr`, see

```
https://en.cppreference.com/w/cpp/language/constexpr
```

Chapter 15, Templates, C++20 Concepts and Metaprogramming, discusses compile-time-programming features.

6.8 Totaling `array` Elements

Often, `array` elements represent a series of values for use in a calculation. For example, if the elements of an `array` represent exam grades, a professor may wish to total the `array` elements and then calculate the class average for the exam. Processing a collection of values into a single value is known as **reduction**—one of the key operations in functional-style programming, which we'll discuss in Section 6.14. Figure 6.5 uses a range-based `for` statement (lines 12–14) to total the values in a four-element `array` of `int`s. Section 6.14 shows how to perform this calculation using the C++ standard library's `accumulate` function.

```
1  // fig06_05.cpp
2  // Compute the sum of an array's elements.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <array>
6
7  int main() {
8     std::array items{10, 20, 30, 40}; // type inferred as array<int, 4>
9     int total{0};
10
11    // sum the contents of items
12    for (const int& item : items) {
13       total += item;
```

```
14      }
15
16      std::cout << fmt::format("Total of array elements: {}\n", total);
17   }
```

```
Total of array elements: 100
```

Fig. 6.5 Compute the sum of an array's elements.

6.9 Using a Primitive Bar Chart to Display array Data Graphically

Many programs present data graphically. For example, numeric values are often visualized in a bar chart, with longer bars representing proportionally larger values. One simple way to display numeric data graphically is with a bar chart that shows each numeric value as a bar of asterisks (`*`).

A professor might graph the number of exam grades in each of several categories to visualize the grade distribution. Suppose the grades were 87, 68, 94, 100, 83, 78, 85, 91, 76 and 87. There was one grade of 100, two grades in the 90s, four in the 80s, two in the 70s, one in the 60s and none below 60. Our next program (Fig. 6.6) stores this data in an `array` of 11 elements, each corresponding to a grade range. For example, element `0` contains the number of grades in the range 0–9, element `7` indicates the number of grades in the range 70–79, and element `10` indicates the number of grades of 100. Upcoming examples will calculate grade frequencies based on a set of values. In this example, we initialize the array `frequencies` with frequency values.

Figure 6.6 reads the numbers from `frequencies` and graphs them as a bar chart, displaying each grade range followed by a bar of asterisks indicating the number of grades in that range. The program operates as follows:

- Line 8 declares the `array` with `constexpr` because the program never modifies `frequencies`, and its values are known at compile-time.
- To label each bar, lines 15–21 output a grade range (e.g., `"70-79: "`) based on the current value of `i`. In lines 16–17, the format specifier `:02d` indicates that an integer (represented by `d`) should be formatted using a field width of `2` and a leading `0` if the integer is fewer than two digits. In line 20, the format specifier `:>5d` indicates that an integer should be right-aligned (`>`) in a field width of `5`.
- The nested `for` statement (lines 26–28) outputs the current bar's asterisks. The loop-continuation condition in line 26 (`stars < frequency`) enables the inner `for` loop to count from `0` up to `frequency`. Thus, a `frequencies` value determines the number of asterisks to display. In this example, `frequencies` elements `0`–`5` contain zeros because no students received a grade below 60. Thus, the program displays no asterisks next to the first six grade ranges.

```
1   // fig06_06.cpp
2   // Printing a student grade distribution as a primitive bar chart.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <array>
6
7   int main() {
8      constexpr std::array frequencies{0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1};
9
10     std::cout << "Grade distribution:\n";
11
12     // for each element of frequencies, output a bar of the chart
13     for (int i{0}; const int& frequency : frequencies) {
14        // output bar labels ("00-09:", ..., "90-99:", "100:")
15        if (i < 10) {
16           std::cout << fmt::format("{:02d}-{:02d}: ",
```

```
17                i * 10, (i * 10) + 9);
18          }
19          else {
20             std::cout << fmt::format("{:>5d}: ",100);
21          }
22
23          ++i;
24
25       // print bar of asterisks
26       for (int stars{0}; stars < frequency; ++stars) {
27          std::cout << '*';
28       }
29
30       std::cout << '\n'; // start a new line of output
31       }
32    }
```

```
Grade distribution:
00-09:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
  100: *
```

Fig. 6.6 Printing a student grade distribution as a primitive bar chart.

6.10 Using array Elements as Counters

Sometimes, programs use counter variables to summarize data, such as a survey's results. Figure 5.3's die-rolling simulation used separate counters to track the frequencies of each die face as the program rolled the die 60,000,000 times. Figure 6.7 reimplements that simulation using an `array` of frequency counters.

```
1   // fig06_07.cpp
2   // Die-rolling program using an array instead of switch.
3   #include <fmt/format.h> // C++20: This will be #include <format>
4   #include <iostream>
5   #include <array>
6   #include <random>
7
8   int main() {
9      // set up random-number generation
10     std::random_device rd; // used to seed the default_random_engine
11     std::default_random_engine engine{rd()}; // rd() produces a seed
12     std::uniform_int_distribution randomDie{1, 6};
13
14     constexpr size_t arraySize{7}; // ignore element zero
15     std::array<int, arraySize> frequency{}; // initialize to 0s
16
17     // roll die 60,000,000 times; use die value as frequency index
18     for (int roll{1}; roll <= 60'000'000; ++roll) {
```

```
19          ++frequency.at(randomDie(engine));
20       }
21
22       std::cout << fmt::format("{}{:>13}\n", "Face", "Frequency");
23
24       // output each array element's value
25       for (size_t face{1}; face < frequency.size(); ++face) {
26          std::cout << fmt::format("{:>4}{:>13}\n", face, frequency.at(face));
27       }
28    }
```

```
Face   Frequency
   1     9997901
   2     9999110
   3    10001172
   4    10003619
   5     9997606
   6    10000592
```

Fig. 6.7 Die-rolling program using an `array` instead of `switch`.

Figure 6.7 uses the `array frequency` (line 15) to count the occurrences of die values. Line 19 replaces the `switch` statement in lines 23–45 of Fig. 5.3. It uses a random die value as an index to determine which `frequency` element to increment for each die roll. The dot (`.`) operator in line 19 has higher precedence than the `++` operator, so the statement selects a `frequency` element, then increments its value.

The call `randomInt(engine)` produces a random index from 1 to 6, so `frequency` must be large enough to store six counters. We use a seven-element `array` in which **we ignore element 0**—it's clearer to have the die value 1 increment `frequency.at(1)` rather than `frequency.at(0)`. So, each face value is used directly as a `frequency` index. We also replace lines 49–54 of Fig. 5.3 by looping through `array frequency` to output the results (Fig. 6.7, lines 25–27). We used a counter-controlled loop here to skip printing element 0 of `frequency`.

6.11 Using arrays to Summarize Survey Results

Our next example uses `array`s to summarize data collected in a survey. Consider the following problem statement:

Twenty students were asked to rate on a scale of 1 to 5 the quality of the food in the student cafeteria, with 1 being "awful" and 5 being "excellent." Place the 20 responses in an `array` of `int` values and determine the frequency of each rating.

This is a popular type of `array`-processing application (Fig. 6.8). We wish to summarize the number of responses of each rating (that is, 1–5). The `array responses` (lines 9–10) `constexpr` because its values do not (and should not) change and are known at compile-time. We use a six-element `array frequency` (line 18) to count each response's number of occurrences. Each `frequency` element is used as a counter for one of the survey responses and is initialized to zero. As in Fig. 6.7, we ignore element 0.

```
1  // fig06_08.cpp
2  // Poll analysis program.
3  #include <fmt/format.h> // C++20: This will be #include <format>
4  #include <iostream>
5  #include <array>
6
7  int main() {
8     // place survey responses in array responses
9     constexpr std::array responses{
10       1, 2, 5, 4, 3, 5, 2, 1, 3, 1, 4, 3, 3, 3, 2, 3, 3, 2, 2, 5};
```

```
11
12      // initialize frequency counters to 0
13      constexpr size_t frequencySize{6}; // size of array frequency
14      std::array<int, frequencySize> frequency{};
15
16      // for each response in responses, use that value
17      // as frequency index to determine element to increment
18      for (const int& response : responses) {
19         ++frequency.at(response);
20      }
21
22      std::cout << fmt::format("{}{:>12}\n", "Rating", "Frequency");
23
24      // output each array element's value
25      for (size_t rating{1}; rating < frequency.size(); ++rating) {
26         std::cout << fmt::format("{:>6}{:>12}\n",
27            rating, frequency.at(rating));
28      }
29   }
```

```
Rating Frequency
     1           3
     2           5
     3           7
     4           2
     5           3
```

Fig. 6.8 Poll analysis program.

The first `for` statement (lines 18–20) takes one response at a time from `responses` (each is a value in the range 1–5) and increments one of the counters `frequency.at(1)` to `frequency.at(5)`. The key statement is line 19, which increments the appropriate counter, depending on the value of `responses.at(response)`. Regardless of the number of responses processed in the survey, the program requires only a six-element `array` (ignoring element zero) to summarize the results because all the response values are between 1 and 5, and the index values for a six-element `array` are 0 through 5.

6.12 Sorting and Searching `array`s

This section uses the built-in C++ standard library **`sort`** function[7] to arrange the elements in an `array` into ascending order and the built-in **`binary_search`** function to determine whether a value is in the `array`.

7. The C++ standard indicates that sort uses an $O(n \log n)$ algorithm, but does not specify which one.

Sorting data—placing it into ascending or descending order—is one of the most important computing applications. Virtually every organization must sort some data and, in many cases, massive amounts of it. Sorting is an intriguing problem that has attracted some of the most intense research efforts in the field of computer science.

Often it may be necessary to determine whether an `array` contains a value that matches a particular **key value**. The process of finding a key value is called **searching**.

Demonstrating Functions `sort` and `binary_search`

Figure 6.9 creates an unsorted `array` of seven `string` objects named `colors` (lines 13–14). The `using` declaration in line 10 enables us to initialize this `array` using **`string` object literals**. You form a `string` object literal by placing an `s` after the string's closing quote. For example, in `"red"s`, the `s` following the literal tells the compiler that this literal is a `std::string` object. Then, in lines 13–14, the compiler can infer from the initializers that the `array`'s element type is `std::string`. Lines 18–20 display `color`'s contents. Note that the loop's control variable is declared as a

```
const std::string&
```

```
1  // fig06_09.cpp
2  // Sorting and searching arrays.
3  #include <array>
4  #include <algorithm> // contains sort and binary_search
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <string>
8
9  int main() {
10    using namespace std::string_literals; // enables string object literals
11
12    // colors is inferred to be an array<string, 7>
13    std::array colors{"red"s, "orange"s, "yellow"s,
14       "green"s, "blue"s, "indigo"s, "violet"s};
15
16    // output original array
17    std::cout << "Unsorted colors array:\n ";
18    for (conststd::string& color : colors) {
19       std::cout << fmt::format("{} ", color);
20    }
21
22    // sort contents of colors
23    std::sort(std::begin(colors), std::end(colors));
24
25    // output sorted array
26    std::cout << "\nSorted colors array:\n ";
27    for (conststd::string& color : colors) {
28       std::cout << fmt::format("{} ", color);
29    }
30
31    // search for "indigo" in colors
32    bool found{std::binary_search(
33       std::begin(colors), std::end(colors), "indigo")};
34    std::cout << fmt::format("\n\n\"indigo\" {} found in colors array\n",
35       found ? "was" : "was not");
36
37    // search for "cyan" in colors
38    found = std::binary_search(
39       std::begin(colors), std::end(colors), "cyan");
```

```
40       std::cout << fmt::format("\"cyan\" {} found in colors array\n",
41          found ? "was" : "was not");
42    }
```

```
Unsorted colors array:
   red orange yellow green blue indigo violet
Sorted colors array:
   blue green indigo orange red violet yellow

"indigo" was found in colors array
"cyan" was not found in colors array
```

Fig. 6.9 Sorting and searching `array`s.

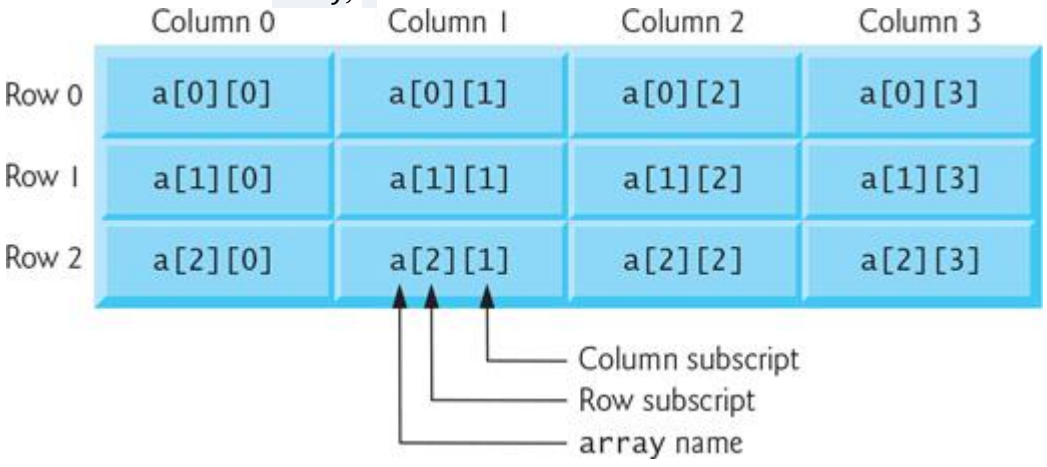
Using a reference for the range variable `color` prevents the loop from copying the current `string` object into `color` during each iteration. In programs processing large numbers of `string`s, that copy operation would degrade performance.

Next, line 23 uses the C++ standard library function **`sort`** (from header `<algorithm>`) to place the elements of the `array colors` into ascending order. For `string`s, this is a **lexicographical sort**—that is, the strings are ordered by their characters' numerical values in the underlying character set. The `sort` function's arguments specify the range of elements to sort—in this case, the entire `array`. The arguments `std::begin(colors)` and `std::end(colors)` return "iterators" that represent the `array`'s beginning and end, respectively. Chapter 13 discusses iterators in depth. Functions **`begin`** and **`end`** are defined in the `<array>` header. As you'll see in later chapters, `sort` can sort the elements of several kinds of data structures. Lines 27–29 display the sorted `array`'s contents.

Err Lines 32–33 and 38–39 use C++ standard library function **`binary_search`** (from header `<algorithm>`) to determine whether a value is in the `array`. **The sequence of values first must be sorted in ascending order**—`binary_search` does not verify this for you. Performing a binary search on an unsorted array is a logic error that could lead to incorrect results. The function's first two arguments represent the range of elements to search, and the third is the search key—the value to find in the `array`. The function returns a `bool` indicating whether the value was found. In Chapter 14, we'll use the C++ Standard function `find` to obtain a search key's index in an `array`.

6.13 Multidimensional **`array`**s

You can use `array`s with two dimensions (i.e., indices) to represent **tables of values** with data arranged in **rows** and **columns**. To identify a particular table element, we must specify two indices. By convention, the first identifies the row, and the second identifies the column. `array`s that require two indices to identify a particular element are called **two-dimensional `array`s** or **2-D `array`s**. `array`s with two or more dimensions are known as **multidimensionalarrays**. The following diagram illustrates a two-dimensional `array`, `a`:



The `array` contains three rows and four columns, so it's said to be a 3-by-4 `array`. In general, an `array` with *m* rows and *n* columns is called an ***m*-by-*n* `array`**.

We have identified every element in the diagram above with an element name of the form `a[`*row*`][`*column*`]`. Similarly, you can access each element with `at`, as in

```
a.at(i).at(j)
```

The elements names in row 0 all have a first index of `0`; the elements names in column 3 all have a second index of `3`.

Figure 6.10 demonstrates initializing two-dimensional `array`s in declarations. Lines 11–12 each create an `array` of `array`s with two rows and three columns.

```
1   // fig06_10.cpp
2   // Initializing multidimensional arrays.
3   #include <iostream>
4   #include <array>
5
6   constexpr size_t rows{2};
7   constexpr size_t columns{3};
8   void printArray(const std::array<std::array<int, columns>, rows>& a);
9
10  int main() {
11     constexpr std::array values1{std::array{1, 2, 3}, std::array{4, 5, 6}};
12     constexpr std::array values2{std::array{1, 2, 3}, std::array{4, 5, 0}};
13
14     std::cout << "values1 by row:\n";
15     printArray(values1);
16
17     std::cout << "\nvalues2 by row:\n";
18     printArray(values2);
19  }
20
21  // output array with two rows and three columns
22  void printArray(const std::array<std::array<int, columns>, rows>& a) {
23     // loop through array's rows
24     for (const auto& row : a) {
25        // loop through columns of current row
26        for (const auto& element : row) {
27           std::cout << element << ' ';
28        }
29
30        std::cout << '\n'; // start new line of output
31     }
32  }
```

```
values1 by row:
1 2 3
4 5 6

values2 by row:
1 2 3
4 5 0
```

Fig. 6.10 Initializing multidimensional `array`s.

Declaring an `array` of `array`s

In lines 11–12, the compiler infers that `values1` and `values2` are `array`s of `array`s with two rows of three columns each and in which the elements are type `int`. Consider the two initializers for `values1`:

```
std::array{1, 2, 3}

std::array{4, 5, 6}
```

From these, the compiler infers that `values1` has two rows. Each of these initializers creates an `array` of three elements, so the compiler infers each row of `values1` has three columns. Finally, the values in these row initializers are `int`s, so the compiler infers that `values1`'s element type is `int`.

Displaying an `array` of `array`s

The program calls function `printArray` to output each `array`'s elements. The function prototype (line 8) and definition (lines 22–32) specify that `printArray` receives a two-row and three-column `array` of `int`s via a `const` reference. In the type

```
const std::array<std::array<int, columns>, rows>&
```

the outer `array` type indicates that it has `rows` (`2`) elements of type

```
array<int, columns>
```

So, each of the outer `array`'s elements is an `array` of `int`s containing `columns` (`3`) elements. The parameter receives the `array` as a `const` reference because `printArray` does not modify the elements.

Nested Range-Based `for` Statements

To process the elements of a two-dimensional `array`, we use a nested loop:

- the outer loop iterates through the rows and
- the inner loop iterates through the columns of a given row.

11 Function `printArray`'s nested loop is implemented with range-based `for` statements. Lines 24 and 26 introduce the C++11 **`auto`** keyword, which tells the compiler to **infer (determine) a variable's data type** based on the variable's initializer value. The outer loop's range variable `row` is initialized with an element from the parameter `a`. Looking at the `array`'s declaration, you can see that it contains elements of type

```
array<int, columns>
```

so the compiler infers that `row` refers to a three-element `array` of `int` values (again, `columns` is 3). The `const&` in `row`'s declaration indicates that the reference cannot be used to modify the rows and prevents each row from being copied into the range variable. The inner loop's range variable `element` is initialized with one element of the `array` represented by `row`. So, the compiler infers that `element` is a reference to a `const int` because each row contains three `int` values. In many IDEs, hovering the mouse cursor over a variable declared with `auto` displays the variable's inferred type. Line 27 displays the `element` value from a given row and column.

Nested Counter-Controlled `for` Statements

We could have implemented the nested loop with counter-controlled iteration as follows:

```
for (size_t row{0}; row < a.size(); ++row) {
   for (size_t column{0}; column < a.at(row).size(); ++column) {
      cout << a.at(row).at(column) << ' '; // or a[row][column]
   }

   cout << '\n';
}
```

Initializing an `array` of `array`s with a Fully Braced Initializer List

If you explicitly specify the `array`'s dimensions when you declare it, you can simplify its initializer list. For example, line 11 could be written as

```
constexpr std::array<std::array<int, columns>, rows> values1{
   {{1, 2, 3}, // row 0
    {4, 5, 6}} // row 1
};
```

In this case, if an initializer sublist has fewer elements than the number of columns, the row's remaining elements would be value initialized.

Common Two-Dimensional `array` Manipulations: Setting a Row's Values

Let's consider several other common manipulations using the three-row and four-column `array` `a` from the diagram at the beginning of Section 6.13. The following `for` statement sets all the elements in row 2 to zero:

```
for (size_t column{0}; column < a.at(2).size(); ++column) {
   a.at(2).at(column) = 0; // or a[2][column]
}
```

The `for` statement varies only the second index (i.e., the column index). The preceding `for` statement is equivalent to the following assignment statements:

```
a.at(2).at(0) = 0; // or a[2][0] = 0;
a.at(2).at(1) = 0; // or a[2][1] = 0;
a.at(2).at(2) = 0; // or a[2][2] = 0;
a.at(2).at(3) = 0; // or a[2][3] = 0;
```

Common Two-Dimensional `array` Manipulations: Totaling All the Elements with Nested Counter-Controlled `for` Loops

The following nested counter-controlled `for` statement totals the elements in the `array a`:

```
int total{0};
for (size_t row{0}; row < a.size(); ++row) {
   for (size_t column{0}; column < a.at(row).size(); ++column) {
      total += a.at(row).at(column); // a[row][column]
   }
}
```

The `for` statement totals the `array`'s elements one row at a time. The outer loop begins by setting the `row` index to `0`, so the elements of row 0 may be totaled by the inner loop. The outer loop then increments `row` to `1`, so the elements of row 1 can be totaled. Then, the outer `for` statement increments `row` to `2`, so the elements of row 2 can be totaled.

Common Two-Dimensional `array` Manipulations: Totaling All the Elements with Nested Range-Based `for` Loops

Nested range-based `for` statements are the preferred way to implement the preceding loop:

```
int total{0};
for (const auto& row : a) { // for each row in a
   for (const auto& column : row) {// for each element in row
      total += column;
   }
}
```

23 C++23 `mdarray`

The C++ Standard Committee is working on a true multidimensional `array` container called `mdarray` for C++23. You can follow the progress on `mdarray` and the related type `mdspan` at

- `mdarray`—https://isocpp.org/files/papers/D1684R0.html
- `mdspan`—https://wg21.link/P0009

6.14 Intro to Functional-Style Programming

Perf Like other popular languages, such as Python, Java and C#, C++ supports several programming paradigms—procedural, object-oriented, generic (template-oriented) and "functional-style." C++'s "functional-style" features help you write more concise code with fewer errors that's easier to read, debug and modify. Functional-style programs also can be easier to parallelize to get better performance on today's multi-core processors.

6.14.1 What vs. How

As a program's tasks get more complicated, the code can become harder to read, debug and modify, and more likely to contain errors. Specifying *how* the code works can become complex. With functional-style programming, you specify *what* you want to do, and library code typically handles "the *how*" for you. This can eliminate many errors.

Consider the following range-based `for` statement from Fig. 6.5, which totals the elements of the `array integers`:

```
for (const int& item : integers) {
   total += item;
}
```

Though this procedural code hides the iteration details, we still have to specify how to total the elements by adding each `item` into the variable `total`. Each time you modify a variable, you can introduce errors. Functional-style programming emphasizes **immutability**—it avoids operations that modify variables' values. If this is your first exposure to functional-style programming, you might be wondering, "How can this be?" Read on.

Functional-Style Reduction with `accumulate`

Figure 6.11 replaces Fig. 6.5's range-based `for` statement with a call to the C++ standard library `accumulate` algorithm (from header `<numeric>`). By default, this function knows *how* to

compute the sum of a range's values, reducing them to a single value. This is known as a **reduction** and is common functional-style programming operation.

```
1  // fig06_11.cpp
2  // Compute the sum of the elements of an array using accumulate.
3  #include <array>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <numeric>
7
8  int main() {
9     constexpr std::array integers{10, 20, 30, 40};
10    std::cout << fmt::format("Total of array elements: {}\n",
11       std::accumulate(std::begin(integers), std::end(integers), 0));
12 }
```

```
Total of array elements: 100
```

Fig. 6.11 Compute the sum of the elements of an `array` using `accumulate`.

Like function `sort` in Section 6.12, function `accumulate`'s first two arguments (line 11) specify the range of elements to sum—in this case, the elements from the beginning to the end of `integers`. The function *internally* adds to the running total of the elements it processes, hiding those calculations. The third argument is the running total's initial value (`0`). You'll see momentarily how to customize `accumulate`'s reduction.

Function `accumulate` uses **internal iteration**, which also is hidden from you. The function knows *how* to iterate through a range of elements and add each element to the running total. Stating *what* you want to do and letting the library determine *how* to do it is known as **declarative programming**—another hallmark of functional programming.

6.14.2 Passing Functions as Arguments to Other Functions: Introducing Lambda Expressions

Many standard library functions allow you to customize how they work by passing other functions as arguments. Functions that receive other functions as arguments are called **higher-order functions** and are used commonly in functional programming. Consider function `accumulate`, which totals elements by default. It also provides an overload, which receives as its fourth argument a function that defines how to perform the reduction. Rather than simply totaling the values, Fig. 6.12 calculates the product of the values.

```
1  // fig06_12.cpp
2  // Compute the product of an array's elements using accumulate.
3  #include <array>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <numeric>
7
8  int multiply(int x, int y) {
9    return x * y;
10 }
11
12 int main() {
13    constexpr std::array integers{1, 2, 3, 4, 5};
14
15    std::cout << fmt::format("Product of integers: {}\n", std::accumulate(
16       std::begin(integers), std::end(integers), 1, multiply));
17
18    std::cout << fmt::format("Product of integers with a lambda: {}\n",
```

```
19          std::accumulate(std::begin(integers), std::end(integers), 1,
20             [](const auto& x, const auto& y){return x * y;}));
21    }
```

```
Product of integers: 120
Product of integers with a lambda: 120
```

Fig. 6.12 Compute the product of an `array`'s elements using `accumulate`.

Calling `accumulate` with a Named Function

Lines 15–16 call `accumulate` for the `array integers` (defined at line 13). We're calculating the product, so the third argument (i.e., the initial value of the reduction) is `1`, rather than `0`; otherwise, the final product would be `0`. The fourth argument is the function to call for every `array` element—in this case, `multiply` (defined in lines 8–10). To calculate a product, this function must receive two arguments:

- the product so far and
- a value from the `array`

and must return a value, which becomes the new product. As `accumulate` iterates through `integers`, it passes the current product and the next element as arguments. For this example, `accumulate` internally calls `multiply` five times:

- The first call passes the initial product (`1`, which was specified as `accumulate`'s third argument) and the array's first element (`1`), producing the product `1`.
- The second call passes the current product (`1`) and the array's second element (`2`), producing the product `2`.
- The third call passes `2` and the array's third element (`3`), producing the product `6`.
- The fourth call passes `6` and the array's fourth element (`4`), producing the product `24`.
- Finally, the last call passes `24` and the array's fifth element (`5`), producing the overall result, `120`, which `accumulate` returns to its caller.

Calling `accumulate` with a Lambda Expression

11 In some cases, you may not need to reuse a function. In such cases, you can define a function where it's needed by using a C++11 **lambda expression** (or simply **lambda**). A lambda expression is essentially an *anonymous function*—that is, a function without a name. The call to `accumulate` in lines 19–20 uses the following lambda expression to perform the same task as `multiply`:

```
[](const auto& x, const auto& y){return x * y;}
```

Lambdas begin with the **lambda introducer** (`[]`), followed by a comma-separated parameter list and a function body. This lambda receives two parameters, calculates their product and returns the result.

14 You saw in Section 6.13 that `auto` enables the compiler to infer a variable's type based on its initial value. Specifying a lambda parameter's type as `auto` enables the compiler to infer the type based on the context in which the lambda is called. In this example, `accumulate` calls the lambda once for each element of the `array`, passing the current product and the element's value as the lambda's arguments. Since the initial product (`1`) is an `int` and the `array` contains `int`s, the compiler infers the lambda parameters' types as `int`. Using `auto` to infer each parameter's type is a C++14 feature called **generic lambdas**. The compiler also infers the lambda's return type from the expression `x * y`—both `x` and `y` are `int`s, so this lambda returns an `int`.

We declared the parameters as `const` references:

- They are `const`, so the lambda's body cannot modify the caller's variables.

- Perf They are references for performance to ensure that if the lambda is used with large objects, it does not copy them.

This lambda could be used with any type that supports the `*` operator. Chapter 14 discusses lambda expressions in detail.

20 6.14.3 Filter, Map and Reduce: Intro to C++20's Ranges Library

The C++ standard library has enabled functional-style programming for many years. C++20's new **ranges library**[8] (header `<ranges>`) makes functional-style programming more convenient. Here, we introduce two key aspects of this library—ranges and views:

8. At the time of this writing, `clang++` does not yet support the ranges library features presented here.

- A **range** is a collection of elements that you can iterate over. So an `array`, for example, is a range.
- A **view** enables you to specify an operation that manipulates a range. Views are **composable**—you can chain them together to process a range's elements through multiple operations.

The program of Fig. 6.13, which we've broken into pieces for discussion purposes, demonstrates several functional-style operations using C++20 ranges. We'll cover more features of this library in Chapter 14.

```
1  // fig06_13.cpp
2  // Functional-style programming with C++20 ranges and views.
3  #include <array>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <numeric>
7  #include <ranges>
8
9  int main() {
10    // lambda to display results of range operations
11    auto showValues{
12       [](auto& values, const std::string& message) {
13          std::cout << fmt::format("{}: ", message);
14
15          for (const auto& value : values) {
16             std::cout << fmt::format("{} ", value);
17          }
18
19          std::cout << '\n';
20       }
21    };
22
```

Fig. 6.13 Functional-style programming with C++20 ranges and views.

`showValues` Lambda for Displaying This Application's Results

Throughout this example, we display various range operations' results using a range-based `for` statement. Rather than repeating that code, we could define a function that receives a range and displays its values. We chose to define a generic lambda (lines 12–20) to show that you can store a lambda in a local variable (`showValues`; lines 11–21). Then, you can use the variable's name to call the lambda, as we do in lines 24, 29, 34, 40 and 51.

Generating a Sequential Range of Integers with `views::iota`

In many of this chapter's examples, we created an `array`, then processed its values. This required preallocating the array with the appropriate number of elements. In some cases, you can generate values *on demand* rather than creating and storing the values in advance. Operations that generate values on demand use **lazy evaluation**, which can reduce your program's memory consumption and improve performance when all the values are not needed at once. Line 23 uses the `<ranges>` library's **`views::iota`** to generate a range of integers from its first argument (`1`) up to, but not including, its second argument (`11`). This is known as a **half-open range**. **The values are not generated until the program iterates over the results**, such as when we call `showValues` (line 24) to display the values.

```
23   auto values1{std::views::iota(1, 11)}; // generate integers 1-10
24   showValues(values1, "Generate integers 1-10");
```

```
Generate integers 1-10: 1 2 3 4 5 6 7 8 9 10
```

Filtering Items with `views::filter`

A common functional-style programming operation is **filtering** elements to select only those that match a condition. This often produces fewer elements than the range being filtered. One way to implement filtering would be a loop that iterates over the elements, using an `if` statement to check whether each element matches a condition. You could then do something with that element, such as add it to a container. That requires explicitly defining an iteration control statement and mutable variables, which can be error-prone, as we discussed in Section 6.6.

With ranges and views, we can use **`views::filter`** to focus on *what* we want to accomplish—in this case, getting the even integers in the range 1–10. The `values2` initializer in lines 27–28 uses the **`|` operator** to connect multiple operations. The first operation (`values1` from line 23) generates `1`–`10`, and the second filters those results. Together, these operations form a **pipeline**. Each pipeline begins with a range, which is the data source (the values `1`–`10` produced by `values1`), followed by an arbitrary number of operations, each separated from the next by `|`.

```
26   // filter each value in values1, keeping only the even integers
27   auto values2{values1 |
28      std::views::filter([](const auto& x) {return x % 2 == 0;})};
29   showValues(values2, "Filtering even integers");
```

```
Filtering even integers: 2 4 6 8 10
```

The argument to `views::filter` must be a function that receives one value to process and returns a `bool` indicating whether to keep the value. We passed a lambda that returns `true` if its argument is divisible by 2.

After lines 27–28 execute, `values2` represents a **lazy pipeline** that can generate the integers 1–10 and filter those values, keeping only the even integers. The pipeline concisely represents *what* we want to do, but not *how* to do it:

- `views::iota` knows *how* to generate integers and
- `views::filter` knows *how* to use its function argument to determine whether to keep each value received from earlier in the pipeline.

However, the pipeline is lazy—no results are produced until you iterate over `values2`.

When `showValues` iterates over `values2`, `views::iota` produces a value, then `views::filter` calls its function argument to determine whether to keep the value. If that function returns `true`, the range-based `for` statement receives that value from the pipeline and displays it. Otherwise, the processing steps repeat with the next value generated by `views::iota`.

Mapping Items with `views::transform`

Another common functional-style programming operation is **mapping** elements to new values, possibly of different types. Mapping produces a result with the same number of elements as the original range being mapped. With C++20 ranges, `views::transform` performs mapping operations. The pipeline in lines 32–33 adds another operation to the pipeline from lines 27–28, mapping the filtered results from `values2` to their squares with the lambda expression passed to `views::transform` in line 33.

```
31   // map each value in values2 to its square
32   auto values3{
33      values2 | std::views::transform([](const auto& x) {return x * x;})};
34   showValues(values3, "Mapping even integers to squares");
```

```
Mapping even integers to squares: 4 16 36 64 100
```

The argument to `views::transform` is a function that receives a value to process and returns the mapped value, possibly of a different type. When `showValues` iterates over the results of the `values3` pipeline:

1. `views::iota` produces a value.
2. `views::filter` determines whether the value is even. If so, the value is passed to *Step 3*. Otherwise, processing proceeds with the next value generated by `views::iota` in *Step 1*.
3. `views:transform` calculates the square of the even integer (as specified by line 33's lambda), then the range-based `for` loop in `showValues` displays the value and processing proceeds with the next value generated by `views::iota` in *Step 1*.

Combining Filtering and Mapping Operations into a Pipeline

A pipeline may contain an arbitrary number of operations separated by `|` operators. The pipeline in lines 37–39 combines all of the preceding operations into a single pipeline, and line 40 displays the results.

```
36   // combine filter and transform to get squares of the even integers
37   auto values4{
38      values1 | std::views::filter([](const auto& x) {return x % 2 == 0;})
39              | std::views::transform([](const auto& x) {return x * x;})};
40   showValues(values4, "Squares of even integers");
```

```
Squares of even integers: 4 16 36 64 100
```

Reducing a Range Pipeline with `accumulate`

C++ standard library functions like `accumulate` also work with lazy range pipelines. Line 44 performs a reduction that sums the squares of the even integers produced by the pipeline in lines 37–39.

```
42   // total the squares of the even integers
```

```
43    std::cout << fmt::format("Sum squares of even integers 2-10: {}\n",
44       std::accumulate(std::begin(values4), std::end(values4), 0));
Sum squares of even integers 2-10: 220
```

Filtering and Mapping an Existing Container's Elements

Various C++ containers, including `array`s and `vector`s (Section 6.15), can be used as the data source in a range pipeline. Line 47 creates an `array` containing `1`–`10`, then uses it in a pipeline that calculates the squares of the even integers in the `array`.

```
46       // process a container's elements
47       constexpr std::array numbers{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
48       auto values5{
49          numbers | std::views::filter([](const auto& x) {return x % 2 == 0;})
50                  | std::views::transform([](const auto& x) {return x * x;})};
51       showValues(values5, "Squares of even integers in array numbers");
52    }
Squares of even integers in array numbers: 4 16 36 64 100
```

6.15 Objects-Natural Case Study: C++ Standard Library Class Template `vector`

We now continue our objects-natural presentation by taking objects of C++ standard library class template **`vector`**[9] (from header `<vector>`) "out for a spin." A `vector` is similar to an `array`, but supports dynamic resizing. At the end of this section, we'll demonstrate `vector`'s bounds checking capabilities, which `array` also has. There, we'll introduce C++'s exception-handling mechanism by detecting and handling an out-of-bounds `vector` index. At that point, we'll discuss the `<stdexcept>` header (line 5). Lines 7–8 are the function prototypes for functions `outputVector` (lines 93–99) and `inputVector` (lines 102–106), which display a `vector`'s contents and input values into a `vector`, respectively.

9. Chapter 13 discusses more `vector` capabilities.

```
1   // fig06_14.cpp
2   // Demonstrating C++ standard library class template vector.
3   #include <iostream>
4   #include <vector>
5   #include <stdexcept>
6
7   void outputVector(const std::vector<int>& items); // display the vector
8   void inputVector(std::vector<int>& items); // input values into the vector
9
```

Fig. 6.14 Demonstrating C++ standard library class template `vector`.

Creating `vector` Objects

Lines 11–12 create two `vector` objects that store values of type `int`—`integers1` contains seven elements, and `integers2` contains 10 elements. By default, all the elements of each `vector` object are set to `0`. Like `array`s, `vector`s can store most data types—simply replace `int` in `std::vector<int>` with the appropriate type.

```
10   int main() {
11      std::vector<int> integers1(7); // 7-element vector<int>
12      std::vector<int> integers2(10); // 10-element vector<int>
```

Note that we used parentheses rather than a braced initializer to pass the size argument to each `vector` object's constructor. When creating a `vector`, if the braces contain one value of the `vector`'s element type, the compiler treats the braces as a one-element initializer list, rather than the `vector`'s size. So the following declaration actually creates a one-element `vector<int>` containing the value `7`, not a seven-element `vector`:

```
std::vector<int> integers1{7};
```

If you know the `vector`'s contents at compile-time, you can use an initializer list and class template argument deduction (CTAD) to define the `vector`, as we've done with `array`'s. For example, the following statement defines a four-element `vector` of `double`s:

```
std::vector integers1{1.1, 2.2, 3.3, 4.4};
```

`vector` Member Function `size`; Function `outputVector`

Line 15 uses `vector` member function **`size`** to obtain `integers1`'s number of elements.[10] Line 17 passes `integers1` to function `outputVector` (lines 93–99), which displays the `vector`'s elements using a range-based `for`. Lines 20 and 22 perform the same tasks for `integers2`.

17 10. You also can use the C++17 global function `std::size`, as in `std::size(integers1)`.

```
14      // print integers1 size and contents
15      std::cout << "Size of vector integers1 is " << integers1.size()
16         << "\nvector after initialization:";
17      outputVector(integers1);
18
19      // print integers2 size and contents
20      std::cout << "\nSize of vector integers2 is " << integers2.size()
21         << "\nvector after initialization:";
22      outputVector(integers2);
23
```

```
Size of vector integers1 is 7
vector after initialization: 0 0 0 0 0 0 0

Size of vector integers2 is 10
vector after initialization: 0 0 0 0 0 0 0 0 0 0
```

Function `inputVector`

Lines 26–27 pass `integers1` and `integers2` to function `inputVector` (lines 102–106) to read values for each `vector`'s elements from the user.

```
24      // input and print integers1 and integers2
25      std::cout << "\nEnter 17 integers:\n";
26      inputVector(integers1);
27      inputVector(integers2);
```

```
28
29       std::cout << "\nAfter input, the vectors contain:\n"
30          << "integers1:";
31       outputVector(integers1);
32       std::cout << "integers2:";
33       outputVector(integers2);
34
```

```
Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1: 1 2 3 4 5 6 7
integers2: 8 9 10 11 12 13 14 15 16 17
```

Comparing `vector` Objects for Inequality

In Chapter 5, we introduced function overloading. A similar concept is operator overloading, which allows you to define how a built-in operator works for a custom type. The C++ standard library already defines overloaded operators `==` and `!=` for `array`s and `vector`s. So, you can compare two `array`s or two `vector`s for equality or inequality, respectively. Line 38 compares two `vector` objects using the `!=` operator. This operator returns `true` if the contents of two `vector`s are not equal—that is, they have different lengths or the elements at the same index in both `vector`s are not equal. Otherwise, `!=` returns `false`.

```
35       // use inequality (!=) operator with vector objects
36       std::cout << "\nEvaluating: integers1 != integers2\n";
37
38       if (integers1 != integers2) {
39          std::cout << "integers1 and integers2 are not equal\n";
40       }
41
```

```
Evaluating: integers1 != integers2
integers1 and integers2 are not equal
```

Initializing One `vector` with the Contents of Another

You can initialize a new `vector` by copying the contents of an existing one. Line 44 creates a `vector` object `integers3` and initializes it with a copy of `integers1`. Here, we used CTAD to infer `integers3`'s element type from that of `integers1`. Line 44 invokes class template `vector`'s copy constructor to perform the copy operation. You'll learn about copy constructors in detail in Chapter 11. Lines 46–48 output the size and contents of `integers3` to demonstrate that it was initialized correctly.

```
42       // create vector integers3 using integers1 as an
43       // initializer; print size and contents
44       std::vector integers3{integers1}; // copy constructor
45
46       std::cout << "\nSize of vector integers3 is " << integers3.size()
47          << "\nvector after initialization: ";
48       outputVector(integers3);
```

```
49
Size of vector integers3 is 7
vector after initialization: 1 2 3 4 5 6 7
```

Assigning `vector`s and Comparing `vector`s for Equality

Line 52 assigns `integers2` to `integers1`, demonstrating that the assignment (`=`) operator is overloaded to work with `vector` objects. Lines 54–57 output the contents of both objects to show that they now contain identical values. Line 62 then compares `integers1` to `integers2` with the equality (`==`) operator to determine whether the contents of the two objects are equal after the assignment, which they are.

```
50      // use overloaded assignment (=) operator
51      std::cout << "\nAssigning integers2 to integers1:\n";
52      integers1 = integers2; // assign integers2 to integers1
53
54      std::cout << "integers1: ";
55      outputVector(integers1);
56      std::cout << "integers2: ";
57      outputVector(integers2);
58
59      // use equality (==) operator with vector objects
60      std::cout << "\nEvaluating: integers1 == integers2\n";
61
62      if (integers1 == integers2) {
63         std::cout << "integers1 and integers2 are equal\n";
64      }
65
```

```
Assigning integers2 to integers1:
integers1: 8 9 10 11 12 13 14 15 16 17
integers2: 8 9 10 11 12 13 14 15 16 17

Evaluating: integers1 == integers2
integers1 and integers2 are equal
```

Using the `at` Member Function to Access and Modify `vector` Elements

Lines 67 and 71 use `vector` member function `at` to obtain an element and use it to get a value from the `vector` and replace a value in the `vector`, respectively. If the index is valid, member function `at` returns either

- a reference to the element at that location, which can be used to change the value of the corresponding `vector` element, or
- a `const` reference to the element at that location, which cannot be used to change the value of the corresponding `vector` element, but can be used to read the element's value.

If `at` is called on a `const vector` or via a reference that's declared `const`, the function returns a `const` reference.

```
66      // use the value at location 5 as an rvalue
67      std::cout << "\nintegers1.at(5) is " << integers1.at(5);
68
69      // use integers1.at(5) as an lvalue
```

```
70      std::cout << "\n\nAssigning 1000 to integers1.at(5)\n";
71      integers1.at(5) = 1000;
72      std::cout << "integers1: ";
73      outputVector(integers1);
74
```

```
integers1.at(5) is 13

Assigning 1000 to integers1.at(5)
integers1: 8 9 10 11 12 1000 14 15 16 17
```

Like `array`s, `vector`s have a `[]` operator. C++ is not required to perform bounds checking when accessing elements with square brackets. Therefore, you must ensure that operations using `[]` do not accidentally manipulate elements outside the `vector`'s bounds.

Exception Handling: Processing an Out-of-Range Index

Line 78 attempts to output the value in `integers1.at(15)`, which is outside the `vector`'s bounds. Member function `at`'s bounds checking recognizes the invalid index and **throws an exception**, which indicates a problem at execution time. The name "exception" suggests that the problem occurs infrequently. **Exception handling** enables you to create **fault-tolerant programs** that can handle (or catch) exceptions. In some cases, this allows a program to continue executing as if no problems were encountered—for example, this program still runs to completion, even though we attempt to access an out-of-range index. More severe problems might prevent a program from continuing normal execution, instead requiring the program to terminate after properly cleaning up any resources it uses (such as closing files, closing database connections, etc.). Here, we introduce exception handling briefly. You'll throw exceptions from your own custom functions in Chapter 9, and we'll discuss exception handling in detail in Chapter 12.

```
75      // attempt to use out-of-range index
76      try {
77         std::cout << "\nAttempt to display integers1.at(15)\n";
78         std::cout << integers1.at(15) << '\n'; // ERROR: out of range
79      }
80      catch (const std::out_of_range& ex) {
81         std::cerr << "An exception occurred: " << ex.what() << '\n';
82      }
83
```

```
Attempt to display integers1.at(15)
An exception occurred: invalid vector subscript
```

The `try` Statement

By default, an exception causes a C++ program to terminate. To handle an exception and possibly enable the program to continue executing, place any code that might throw an exception in a **`try` statement** (lines 76–82). The **`try` block** (lines 76–79) contains the code that might throw an exception, and the **`catch` block** (lines 80–82) contains the code that handles the exception if one occurs. As you'll see in Chapter 12, a single `try` block can have many `catch` blocks to handle different types of exceptions that might be thrown. If the code in the `try` block executes successfully, lines 80–82 are ignored. The braces that delimit `try` and `catch` blocks' bodies are required.

Executing the `catch` Block

When the program calls `vector` member function `at` with the argument `15` (line 78), the function attempts to access the element at location 15, which is outside the `vector`'s bounds—`integers1` has only 10 elements at this point. Because bounds checking is performed at execution time, `vector` member function `at` generates an exception. Specifically, line 78 throws an **`out_of_range`** exception (from header `<stdexcept>`) to notify the program of this problem. This immediately terminates the `try` block, skipping any remaining statements in that block. Then, the `catch` block begins executing. If you declared any variables in the `try` block, they're now out of scope and not accessible to the `catch` block.

A `catch` block should declare its exception parameter (`ex`) as a `const` reference—we'll say more about this in Chapter 12. The `catch` block can handle exceptions of the specified type. Inside the block, you can use the parameter's identifier to interact with a caught exception object.

`what` Member Function of the Exception Object

When lines 80–82 catch the exception, the program displays a message (which varies by compiler) indicating the problem that occurred. Line 81 calls the exception object's **`what`** member function to get the error message. Once the message is displayed in this example, the exception is considered handled, and the program continues with the next statement after the `catch` block's closing brace. In this example, lines 85–89 execute next.

Changing the Size of a `vector`

One key difference between a `vector` and an `array` is that a `vector` can dynamically grow and shrink as its number of elements varies. To demonstrate this, line 85 shows the current size of `integers3`, line 86 calls the `vector`'s **`push_back`** member function to add a new element containing 1000 to the end of the `vector`, and line 87 shows the new size of `integers3`. Line 89 then displays `integers3`'s new contents.

```
84       // changing the size of a vector
85       std::cout << "\nCurrent integers3 size is: " << integers3.size();
86       integers3.push_back(1000); // add 1000 to the end of the vector
87       std::cout << "\nNew integers3 size is: " << integers3.size()
88          << "\nintegers3 now contains: ";
89       outputVector(integers3);
90    }
91
```

```
Current integers3 size is: 7
New integers3 size is: 8
integers3 now contains: 1 2 3 4 5 6 7 1000
```

Functions `outputVector` and `inputVector`

Function `outputVector` uses a range-based `for` statement to obtain the value in each element of the `vector` for output. As with class template `array`, you could also do this using a counter-controlled loop, but the range-based `for` is recommended. Similarly, function `inputVector` uses a range-based `for` statement with an `int&` range variable that can be used to store an input value in the corresponding `vector` element.

```
92    // output vector contents
93    void outputVector(const std::vector<int>& items) {
94       for (const int& item : items) {
```

```
95          std::cout << item << " ";
96       }
97
98       std::cout << '\n';
99    }
100
101   // input vector contents
102   void inputVector(std::vector<int>& items) {
103      for (int& item : items) {
104         std::cin >> item;
105      }
106   }
```

Straight-Line Code

As you worked through this chapter's examples, you saw lots of `for` loops. But one thing you may have noticed in our Objects-Natural sections is that much of the code for creating and using objects is straight-line sequential code with few control statements. Working with objects often "flattens" code into lots of sequential function calls.

6.16 Wrap-Up

This chapter began our introduction to data structures, exploring the use of C++ standard library class templates `array` and `vector` to store data in and retrieve data from lists and tables of values. We demonstrated how to declare an `array`, initialize an `array` and refer to individual elements of an `array`. We passed `array`s to functions by reference and used the `const` qualifier to prevent the called function from modifying the `array`'s elements. You learned how to use C++11's range-based `for` statement to manipulate all the elements of an `array`. We also demonstrated C++ standard library functions `sort` and `binary_search` to sort an unsorted `array` and search a sorted `array`, respectively.

You learned how to declare and manipulate two-dimensional `array`s. We used nested counter-controlled and nested range-based `for` statements to iterate through all the rows and columns of a two-dimensional `array`. We also showed how to use `auto` to infer a variable's type based on its initializer value. We introduced functional-style programming in C++ using C++20 ranges and views to compose lazy pipelines of operations.

In this chapter's Objects-Natural case study, we demonstrated the capabilities of the C++ standard library class template `vector`. In that example, we discussed how to access `array` and `vector` elements with bounds checking and demonstrated basic exception-handling concepts. In later chapters, we'll continue our coverage of data structures.

In Chapter 7, we present one of C++'s most powerful features—the pointer. Pointers keep track of where data is stored in memory, which allows us to manipulate those items in interesting ways. As you'll see, C++ also provides a language element called an array (different from the class template `array`) that's closely related to pointers. In contemporary C++ code, it's considered better practice to use the `array` class template rather than traditional arrays.

7. (Downplaying) Pointers in Modern C++

Objectives

In this chapter, you'll:

- Learn what pointers are, and how to declare and initialize them.
- Use the address (`&`) and indirection (`*`) pointer operators.
- Compare the capabilities of pointers and references.
- Use pointers to pass arguments to functions by reference.
- Use pointer-based arrays and strings mostly in legacy code.
- Use `const` with pointers and the data they point to.
- Use operator `sizeof` to determine the number of bytes that store a value of a particular type.
- Understand pointer expressions and pointer arithmetic that you'll see in legacy code.
- Use C++11's `nullptr` to represent pointers to nothing.
- Use C++11's `begin` and `end` library functions with pointer-based arrays.
- Learn various C++ Core Guidelines for avoiding pointers and pointer-based arrays to create safer, more robust programs.
- Use C++20's `to_array` function to convert built-in arrays and initializer lists to `std::array`s.
- Continue our Objects-Natural approach by using C++20's class template `span` to create objects that are views into built-in arrays, `std::array`s and `std::vector`s.
-

Outline

7.1 Introduction
7.2 Pointer Variable Declarations and Initialization
7.2.1 Declaring Pointers
7.2.2 Initializing Pointers
7.2.3 Null Pointers Before C++11
7.3 Pointer Operators
7.3.1 Address (`&`) Operator
7.3.2 Indirection (`*`) Operator
7.3.3 Using the Address (`&`) and Indirection (`*`) Operators
7.4 Pass-by-Reference with Pointers
7.5 Built-In Arrays
7.5.1 Declaring and Accessing a Built-In Array
7.5.2 Initializing Built-In Arrays
7.5.3 Passing Built-In Arrays to Functions
7.5.4 Declaring Built-In Array Parameters
7.5.5 C++11 Standard Library Functions `begin` and `end`
7.5.6 Built-In Array Limitations
7.6 Using C++20 `to_array` to convert a Built-in Array to a `std::array`
7.7 Using `const` with Pointers and the Data Pointed To
7.7.1 Using a Nonconstant Pointer to Nonconstant Data
7.7.2 Using a Nonconstant Pointer to Constant Data
7.7.3 Using a Constant Pointer to Nonconstant Data
7.7.4 Using a Constant Pointer to Constant Data
7.8 `sizeof` Operator
7.9 Pointer Expressions and Pointer Arithmetic
7.9.1 Adding Integers to and Subtracting Integers from Pointers

7.9.2 Subtracting One Pointer from Another
7.9.3 Pointer Assignment
7.9.4 Cannot Dereference a `void*`
7.9.5 Comparing Pointers
7.10 Objects-Natural Case Study: C++20 `span`s—Views of Contiguous Container Elements
7.11 A Brief Intro to Pointer-Based Strings
7.11.1 Command-Line Arguments
7.11.2 Revisiting C++20's `to_array` Function
7.12 Looking Ahead to Other Pointer Topics
7.13 Wrap-Up

7.1 Introduction

This chapter discusses pointers, built-in pointer-based arrays and pointer-based strings (also called C-strings), each of which C++ inherited from the C programming language.

Downplaying Pointers in Modern C++

Pointers are powerful but challenging to work with and error-prone. So, Modern C++ (C++20, C++17, C++14 and C++11) has added features that eliminate the need for most pointers. New software-development projects generally should prefer:

- using references or "smart pointer" objects (Section 11.5) over pointers,
- using `std::array` and `std::vector` objects (Chapter 6) over using built-in pointer-based arrays and
- using `std::string` objects (Chapters 2 and 8) and `std::string_view` objects (Section 8.11) over pointer-based C-strings.

Sometimes Pointers Are Still Required

You'll encounter pointers, pointer-based arrays (also called C-style arrays) and pointer-based C-strings frequently in the massive installed base of legacy C++ code. Pointers are required to:

- create and manipulate dynamic data structures, like linked lists, queues, stacks and trees that can grow and shrink at execution time—though most programmers will use the C++ standard library's existing dynamic containers like `vector` and the containers we discuss in Chapter 13,
- process command-line arguments, which a program receives as a pointer-based array of C-strings, and
- pass arguments by reference if there's a possibility of a `nullptr`[1] (i.e., a pointer to nothing; Section 7.2.2)—a reference must refer to an actual object.[2]

1. C++ Core Guidelines, "F.60: Prefer `T*` over `T&` When "No Argument" Is a Valid Option." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-ptr-ref.

2. In modern C++, you can represent the presence or absence of an object is std::optional (C++17; https://en.cppreference.com/w/cpp/utility/optional).

Pointer-Related C++ Core Guidelines

CG We mention C++ Core Guidelines that encourage you to make your code safer and more robust by using techniques that avoid pointers, pointer-based arrays and pointer-based strings. For example, several guidelines recommend implementing pass-by-reference using references rather than pointers.[3]

---

3. C++ Core Guidelines, "F: Functions." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-functions.

C++20 Features for Avoiding Pointers

20 For programs that still require pointer-based arrays (e.g., command-line arguments), C++20 adds two new features that help make your programs safer and more robust:

- Function `to_array` converts a pointer-based array to a `std::array`, so you can take advantage of the features we demonstrated in Chapter 6.
- `span`s offer a safer way to pass built-in arrays to functions. They're iterable, so you can use them with range-based `for` statements to conveniently process elements without risking out-of-bounds array accesses. Also, because `span`s are iterable, you can use them with standard library container-processing algorithms, such as `accumulate` and `sort`. In this chapter's Objects-Natural case study (Section 7.10), you'll see that `span`s also work with `std::array` and `std::vector`.

The key takeaway from reading this chapter is to avoid using pointers, pointer-based arrays and pointer-based strings whenever possible. If you must use them, take advantage of `to_array` and `span`s.

Other Concepts Presented in This Chapter

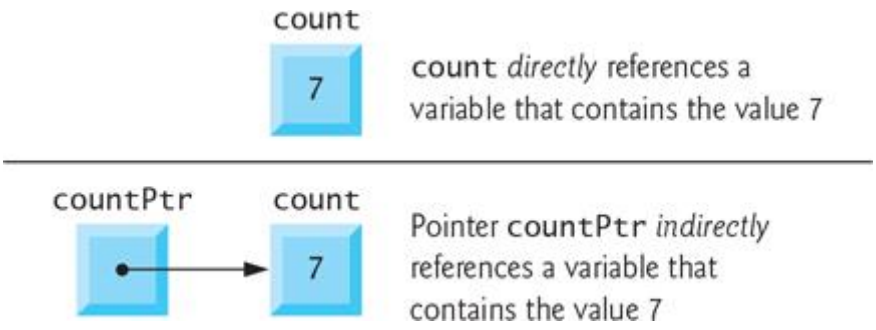
We declare and initialize pointers and demonstrate the pointer operators `&` and `*`. In Chapter 5, we performed pass-by-reference with references. Here, we show that pointers also enable pass-by-reference. We demonstrate built-in, pointer-based arrays and their intimate relationship with pointers.

We show how to use `const` with pointers and the data they point to, and we introduce the `sizeof` operator to determine the number of bytes that store values of particular fundamental types and pointers. We demonstrate pointer expressions and pointer arithmetic.

C-strings were used widely in older C++ software. This chapter briefly introduces C-strings. You'll see how to process command-line arguments—a simple task for which C++ still requires you to use both C-strings and pointer-based arrays.

7.2 Pointer Variable Declarations and Initialization

Pointer variables contain memory addresses as their values. Usually, a variable directly contains a specific value. A pointer contains the memory address of a variable that, in turn, contains a specific value. In this sense, a variable name **directly references a value**, and a pointer **indirectly references a value**, as shown in the following diagram:



Referencing a value through a pointer is called **indirection**.

7.2.1 Declaring Pointers

The following declaration declares the variable `countPtr` to be of type `int*` (i.e., a pointer to an `int` value) and is read (right-to-left) as "`countPtr` is a pointer to an `int`":

```
int* countPtr{nullptr};
```

This `*` is not an operator; rather, it indicates that the variable to its right is a pointer. We like to include the letters `Ptr` in each pointer variable name to make it clear that the variable is a pointer and must be handled accordingly.

7.2.2 Initializing Pointers

11 Sec Initialize each pointer with `nullptr` (from C++11) or a memory address. A pointer with the value `nullptr` "points to nothing" and is known as a **null pointer**. From this point forward, when we refer to a "null pointer," we mean a pointer with the value `nullptr`. Initialize all pointers to prevent pointing to unknown or uninitialized areas of memory.

7.2.3 Null Pointers Before C++11

CG In legacy C++, null pointers were specified by `0` or `NULL`. `NULL` is defined in several standard library headers to represent the value `0`. Initializing a pointer with `NULL` is equivalent to initializing it to `0`. For modern C++, the C++ Core Guidelines indicate you should always use `nullptr` rather than `0` or `NULL`.[4]

4. C++ Core Guidelines, "ES.47: Use nullptr Rather Than 0 or NULL." Accessed December 31, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-nullptr.

7.3 Pointer Operators

The unary operators `&` and `*` create pointer values and "dereference" pointers, respectively. We show how to use these operators in the following sections.

7.3.1 Address (&) Operator

The **address operator (&)** is a unary operator that obtains the memory address of its operand. For example, assuming the declarations

```
int y{5}; // declare variable y
int* yPtr{nullptr}; // declare pointer variable yPtr
```

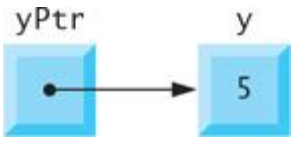
the following statement assigns the address of the variable `y` to pointer variable `yPtr`:

```
yPtr = &y; // assign address of y to yPtr
```

Variable `yPtr` is said to "point to" `y`. Now, `yPtr` indirectly references the variable `y`'s value.

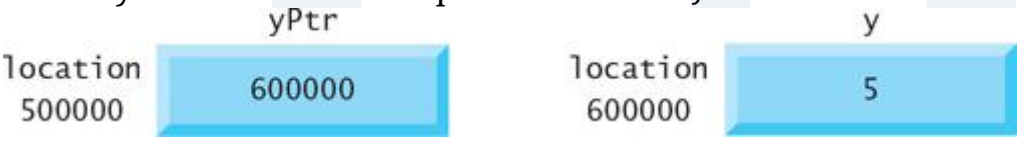
The `&` in the preceding statement is not a reference variable declaration—in that context, `&` would be preceded by a type name and the `&` is part of the type. In an expression like `&y`, the `&` is the address operator.

The following diagram shows a memory representation after the previous assignment:



The "pointing relationship" is indicated by drawing an arrow from the box representing the pointer `yPtr` in memory to the box representing the variable `y` in memory.

The following diagram shows another pointer memory representation with `int` variable `y` stored at memory location `600000` and pointer variable `yPtr` at location `500000`.



The address operator's operand must be an *lvalue*—the address operator cannot be applied to literals or to expressions that result in temporary values (like the results of calculations).

7.3.2 Indirection (*) Operator

Applying the unary `*` **operator** to a pointer results in an *lvalue* representing the object to which its pointer operand points. This operator is known as the **indirection operator** or **dereferencing operator**. If `yPtr` points to `y` and `y` contains `5` (as in the preceding diagrams), the statement

```
std::cout << *yPtr << '\n';
```

displays `y`'s value (`5`), as would the statement

```
std::cout << y << '\n';
```

Using `*` in this manner is called **dereferencing a pointer**. A dereferenced pointer also can be used as an *lvalue* in an assignment. The following assigns `9` to `y`:

```
*yPtr = 9;
```

In this statement, `*yPtr` is an *lvalue* referring to `y`. The dereferenced pointer may also be used to receive an input value as in

```
cin >> *yPtr;
```

which places the input value in `y`.

Undefined Behaviors

Sec Err Dereferencing an uninitialized pointer or a pointer that no longer points to an object results in undefined behavior that could cause a fatal runtime error. This also could lead to accidentally modifying important data, allowing the program to run to completion, possibly with incorrect results. An attacker might exploit this potential security flaw to access data, overwrite data or even execute malicious code.[5,6,7] Using the result of dereferencing a null pointer or a pointer that no longer points to an object is undefined behavior that often causes a fatal execution-time error. If you must use pointers, ensure that a pointer is not `nullptr` before dereferencing it.

---

5. "Undefined Behavior." Wikipedia. Wikimedia Foundation. Accessed December 31, 2021. https://en.wikipedia.org/wiki/Undefined_behavior.

6. "Common Weakness Enumeration." CWE. Accessed January 2, 2022. https://cwe.mitre.org/data/definitions/824.html.

7. "Dangling Pointer." Wikipedia. Wikimedia Foundation. Accessed December 31, 2021. https://en.wikipedia.org/wiki/Dangling_pointer.

7.3.3 Using the Address (&) and Indirection (*) Operators

Figure 7.1 demonstrates the `&` and `*` pointer operators, which have the third-highest precedence. See Appendix A for the complete operator-precedence chart. Memory locations are output by `<<` in this example as hexadecimal (i.e., base-16) integers. (See online Appendix C, Number Systems, for more information on hexadecimal integers.) The output shows that variable `a`'s address (line 9) and `aPtr`'s value (line 10) are identical, confirming that `a`'s address was indeed assigned to `aPtr` (line 7). The outputs from lines 11–12 confirm that `*aPtr` has the same value as `a`. The displayed memory addresses are compiler- and platform-dependent. They typically change with each program execution, so you'll likely see different addresses.

```
1  // fig07_01.cpp
2  // Pointer operators & and *.
3  #include <iostream>
4
5  int main() {
6     constexpr int a{7}; // initialize a with 7
```

```
7     const int* aPtr{&a}; // initialize aPtr with address of int variable a
8
9     cout << "The address of a is " << &a
10       << "\nThe value of aPtr is " << aPtr;
11    cout << "\n\nThe value of a is " << a
12       << "\nThe value of *aPtr is " << *aPtr << '\n';
13 }
```

```
The address of a is  000000D649EFFD00
The value of aPtr is  000000D649EFFD00

The value of a is 7
The value of *aPtr is 7
```

Fig. 7.1 Pointer operators `&` and `*`.

7.4 Pass-by-Reference with Pointers

There are three ways in C++ to pass arguments to a function:

- pass-by-value,
- pass-by-reference with a reference argument and
- **pass-by-reference with a pointer argument** (sometimes called pass-by-pointer).

Perf Chapter 5 showed the first two. Here, we explain pass-by-reference with a pointer. Pointers, like references, can be used to modify variables in the caller or pass large data objects by reference to avoid the overhead of copying objects. When calling a function that receives a pointer, pass a variable's address by applying the address operator (`&`) to the variable's name.

An Example of Pass-By-Value

Figures 7.2 and 7.3 present two functions that each cube an integer. Figure 7.2 passes variable `number` by value (line 12) to function `cubeByValue` (lines 17–19), which cubes its argument and passes the result back to `main` using a `return` statement (line 18).[8] We store the new value in `number` (line 12), overwriting its original value.

8. We also could calculate the cube of `n` with `std::pow(n, 3)`.

```
1  // fig07_02.cpp
2  // Pass-by-value used to cube a variable's value.
3  #include <fmt/format.h>
4  #include <iostream>
5
6  int cubeByValue(int n); // prototype
7
8  int main() {
9     int number{5};
10
11    std::cout << fmt::format("Original value of number is {}\n", number);
12    number = cubeByValue(number); // pass number by value to cubeByValue
13    std::cout << fmt::format("New value of number is {}\n", number);
14 }
15
16 // calculate and return cube of integer argument
17 int cubeByValue(int n) {
```

```
18      return n * n * n; // cube local variable n and return result
19   }
Original value of number is 5
New value of number is 125
```

Fig. 7.2 Pass-by-value used to cube a variable's value.

An Example of Pass-By-Reference with Pointers

Figure 7.3 passes the variable `number` to function `cubeByReference` using pass-by-reference with a pointer argument (line 13)—the address of `number` is passed to the function. Function `cubeByReference` (lines 18–20) specifies parameter `nPtr` (a pointer to `int`) to receive its argument. The function uses the dereferenced pointer—`*nPtr`, an alias for `number` in `main`—to cube the value to which `nPtr` points (line 19). This directly changes the value of `number` in `main` (line 10). Line 19 can be made clearer with redundant parentheses:

```
*nPtr = (*nPtr) * (*nPtr) * (*nPtr); // cube *nPtr
```

```
 1  // fig07_03.cpp
 2  // Pass-by-reference with a pointer argument used to cube a
 3  // variable's value.
 4  #include <fmt/format.h>
 5  #include <iostream>
 6
 7  void cubeByReference(int* nPtr); // prototype
 8
 9  int main() {
10     int number{5};
11
12     std::cout << fmt::format("Original value of number is {}\n", number);
13     cubeByReference(&number); // pass number address to cubeByReference
14     std::cout << fmt::format ("New value of number is {}\n", number);
15  }
16
17  // calculate cube of *nPtr; modifies variable number in main
18  void cubeByReference(int* nPtr) {
19     *nPtr = *nPtr * *nPtr * *nPtr; // cube *nPtr
20  }
Original value of number is 5
New value of number is 125
```

Fig. 7.3 Pass-by-reference with a pointer argument used to cube a variable's value.

A function receiving an address as an argument must define a pointer parameter to receive the address. For example, function `cubeByReference`'s header (line 18) specifies that the function receives a pointer to an `int` as an argument, stores the address in `nPtr` and does not return a value.

Insight: Pass-By-Reference with a Pointer Actually Passes the Pointer By Value

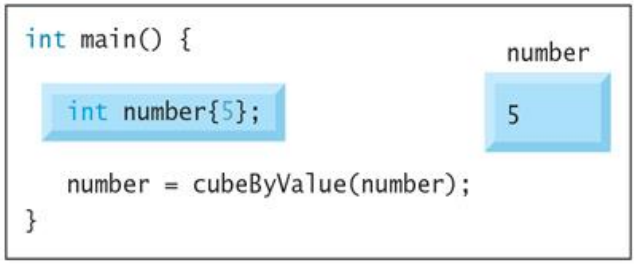
Passing a variable by reference with a pointer does not actually pass anything by reference. Instead, a pointer to that variable is passed by value. That pointer value is copied into the function's

corresponding pointer parameter. The called function can then dereference the pointer to access the caller's variable, thus accomplishing pass-by-reference.

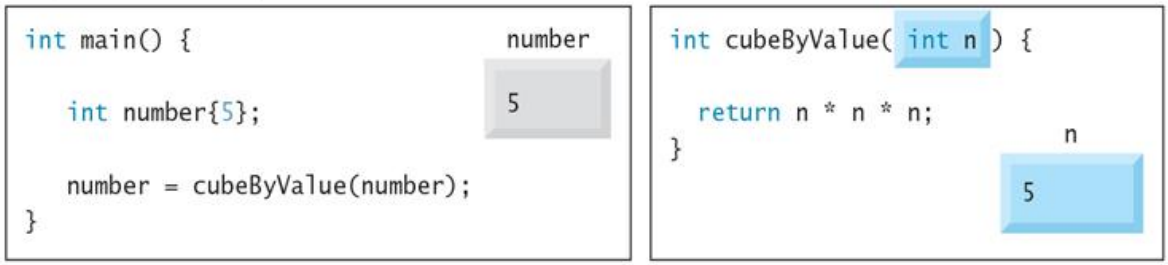
Graphical Analysis of Pass-By-Value and Pass-By-Reference

Figures 7.4 and 7.5 graphically analyze the execution of Figs. 7.2 and 7.3, respectively. The rectangle above a given expression or variable contains the value produced by a step in the diagram. Each diagram's right column shows functions `cubeByValue` (Fig. 7.2) and `cubeByReference` (Fig. 7.3) only when they're executing.

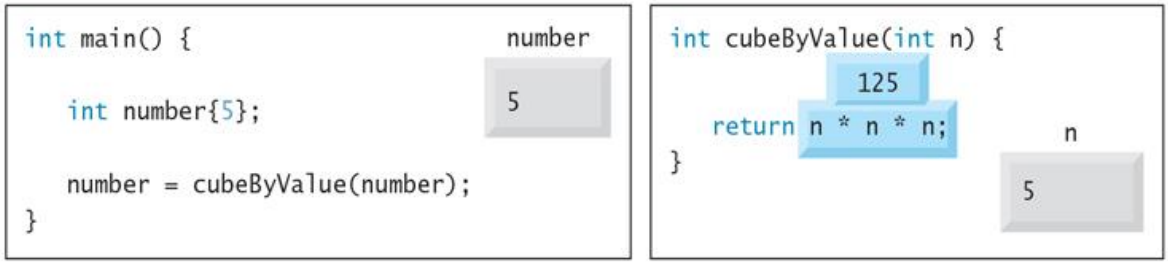
Step 1: Before `main` calls `cubeByValue`:



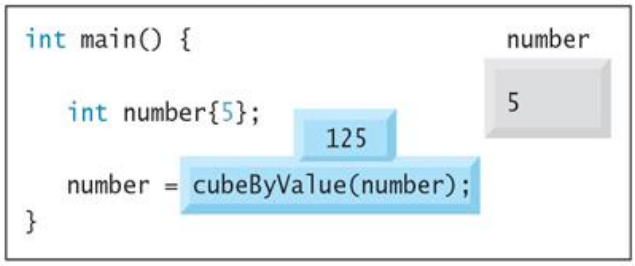
Step 2: After `cubeByValue` receives the call:



Step 3: After `cubeByValue` cubes parameter `n` and before `cubeByValue` returns to `main`:



Step 4: After `cubeByValue` returns to `main` and before assigning the result to `number`:



Step 5: After `main` completes the assignment to `number`:

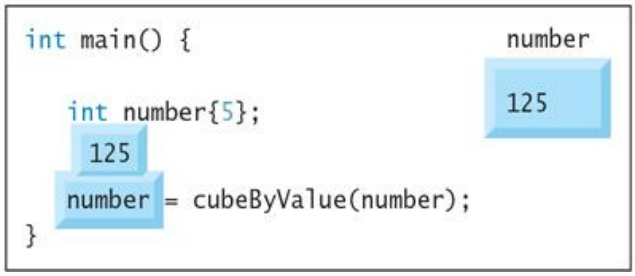
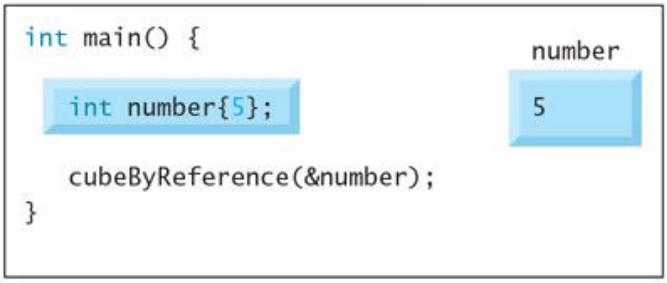
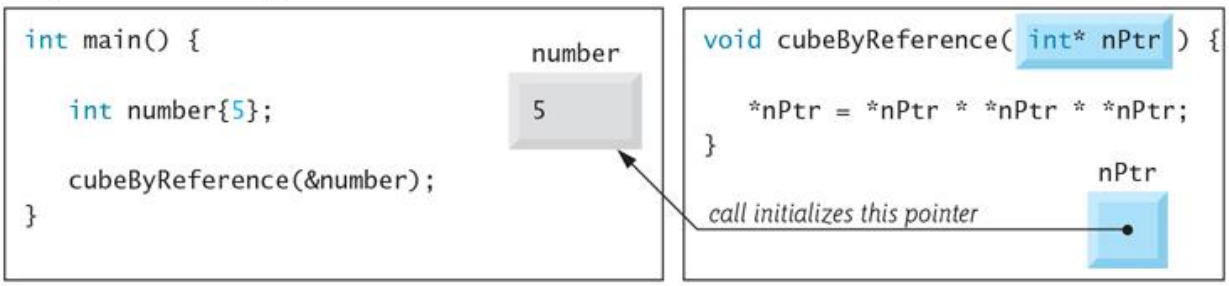


Fig. 7.4 Pass-by-value analysis of the program of Fig. 7.2.

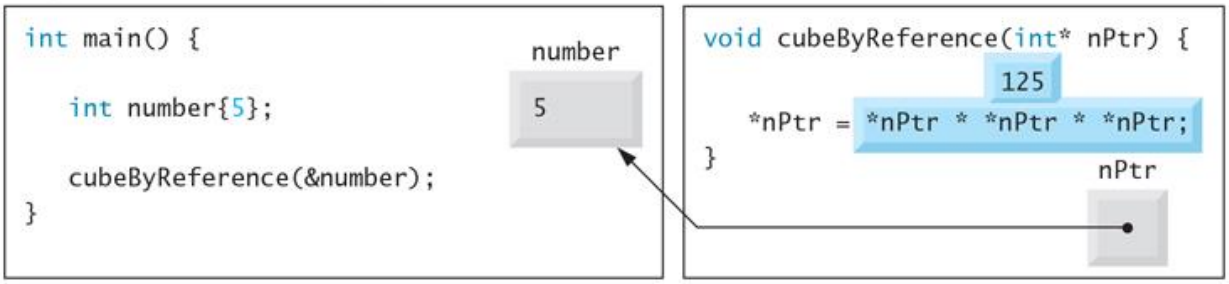
Step 1: Before `main` calls `cubeByReference`:



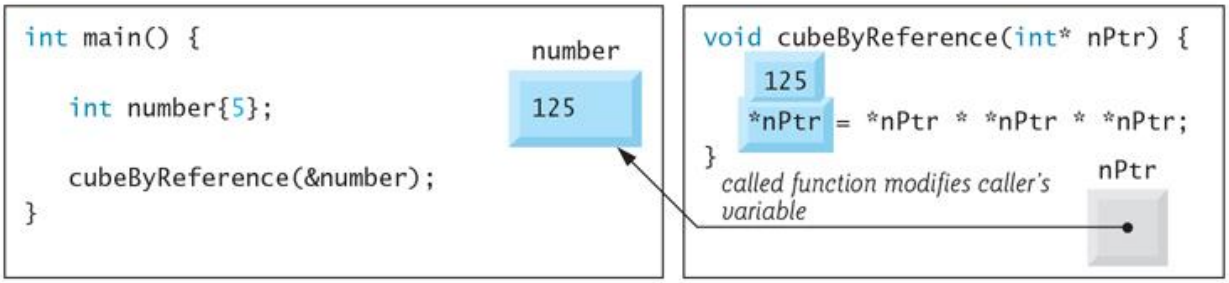
Step 2: After `cubeByReference` receives the call and before `*nPtr` is cubed:



Step 3: Before `*nPtr` is assigned the result of the calculation `5 * 5 * 5`:



Step 4: After `*nPtr` is assigned `125` and before program control returns to `main`:



Step 5: After `cubeByReference` returns to `main`:

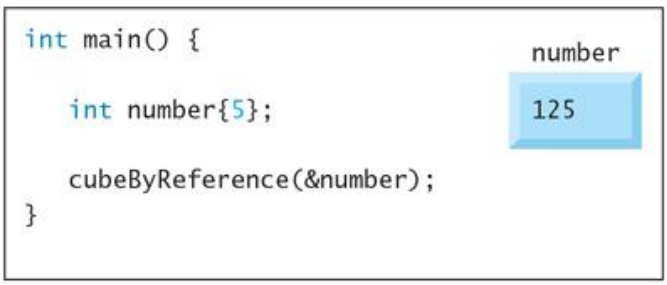


Fig. 7.5 Pass-by-reference analysis of the program of Fig. 7.3.

7.5 Built-In Arrays

Sec Here we present built-in arrays, which like `std::array`s, are fixed-size data structures. We include this presentation mostly because you'll see built-in arrays in legacy C++ code. **New applications generally should use `std::array` and `std::vector` to create safer, more robust applications.** 20 20 In particular, `std::array` and `std::vector` objects always know their own size—even when passed to other functions, which is not the case for built-in arrays. If you work on applications containing built-in arrays, you can use C++20's `to_array` function to convert them to `std::array`s (Section 7.6), or you can process them more safely using C++20's `span`s (Section 7.10). There are some cases in which built-in arrays are required, such as when processing command-line arguments (Section 7.11).

7.5.1 Declaring and Accessing a Built-In Array

As with `std::array`, you must specify a built-in array's element type and number of elements, but the syntax is different. For example, to reserve five elements for a built-in array of `int`s named `c`, use

```
int c[5]; // c is a built-in array of 5 integers
```

You use the subscript (`[]`) operator to access a built-in array's elements. Recall from Chapter 6 that the subscript (`[]`) operator does not provide bounds checking for `std::array`s—this is also true for built-in arrays. Of course, `std::array`'s `at` member function does bounds checking. Built-in arrays do not have a range-checked `at` indexing function, but you can use the Guidelines Support Library (GSL) function `gsl::at`, which receives as arguments the array and an index. **This function's first argument must be the array's name, not a pointer to the array's first element.**

7.5.2 Initializing Built-In Arrays

You can initialize the elements of a built-in array using an initializer list. For example,

```
int n[5]{50, 20, 30, 10, 40};
```

creates and initializes a built-in array of five `int`s. If you provide fewer initializers than the number of elements, the remaining elements are **value initialized**—fundamental numeric types are set to `0`, `bool`s are set to `false`, pointers are set to `nullptr` and objects receive the default initialization specified by their class definitions (Chapter 9). If you provide too many initializers, a compilation error occurs. The compiler can size a built-in array by counting an initializer list's elements. For example, the following creates a five-element `array`:

```
int n[]{50, 20, 30, 10, 40};
```

7.5.3 Passing Built-In Arrays to Functions

The value of a built-in array's name is implicitly convertible to a `const` or non-`const` pointer to the built-in array's first element—this is known as **decaying to a pointer**. So the array name `n` above is implicitly convertible to `&n[0]`, which is a pointer to the element containing `50`. You don't need to take a built-in array's address (`&`) to pass it to a function—you simply pass its name. As you saw in Section

7.4, a function that receives a pointer to a variable in the caller can modify that variable in the caller. For built-in arrays, the called function can modify all the elements in the caller—unless the parameter is declared `const` to prevent the argument array in the caller from being modified.

7.5.4 Declaring Built-In Array Parameters

You can declare a built-in array parameter in a function header, as follows:

```
int sumElements(const int values[], size_t numberOfElements)
```

Here, the function's first argument should be a one-dimensional built-in array of `const int`s. Unlike `std::array`s and `std::vector`s, built-in arrays don't know their own size, so a function that processes a built-in array also should receive the built-in array's size.

The preceding header also can be written as

```
int sumElements(const int* values, size_t numberOfElements)
```

The compiler does not differentiate between a function that receives a pointer and a function that receives a built-in array. In fact, the compiler converts `const int values[]` to `const int* values` under the hood. This means the function must "know" when it's receiving a pointer to the first element of a built-in array vs. a single variable that's being passed by reference.

CG 20 **The C++ Core Guidelines say not to pass built-in arrays to functions as just a pointer.[9] Instead, you should pass C++20 `span`s because they maintain a pointer to the array's first element and its size.** In Section 7.10, we'll demonstrate `span`s, and you'll see that passing a `span` is superior to passing a built-in array and its size to a function.

---

9. C++ Core Guidelines, "I.13: Do not pass an array as a single pointer." Accessed January 2, 2021. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array.

11 7.5.5 C++11 Standard Library Functions `begin` and `end`

In Section 6.12, we sorted a `std::array` of `string`s called `colors` as follows:

```
std::sort(std::begin(colors), std::end(colors));
```

Functions `begin` and `end` specified that the entire `std::array` should be sorted. Function `sort` (and many other C++ standard library functions) also can be applied to built-in arrays. For example, to sort the built-in array `n` (Section 7.5.2), you can write

```
std::sort(std::begin(n), std::end(n)); // sort built-in array n
```

20 **For a built-in array, `begin` and `end` work only with the array's name, not with a pointer to the array's first element.** Again, you should pass built-in arrays to other functions using C++20 `span`s, which we demonstrate in Section 7.10.

7.5.6 Built-In Array Limitations

Built-in arrays have several limitations:

- They cannot be compared using the relational and equality operators. You must use a loop to compare two built-in arrays element by element. If you had two `int` arrays named `array1` and `array2`, the condition `array1 == array2` would always be false, even if the arrays' contents are identical. Remember, array names decay to `const` pointers to the arrays' first elements. And, of course, for separate arrays, those elements reside at different memory locations.
- The elements of a built-in array cannot be assigned all at once to another built-in array with a simple assignment, as in `builtInArray1 = builtInArray2`.
- They don't know their own size, so a function that processes a built-in array typically requires as arguments both the built-in array's name and its size.
- They don't provide automatic bounds checking—you must ensure that array-access expressions use subscripts within the built-in array's bounds.

20 7.6 Using C++20 `to_array` to Convert a Built-in Array to a `std::array`

CG Sec In industry, you'll encounter C++ legacy code that uses built-in arrays. The C++ Core Guidelines say you should prefer `std::array`s and `std::vector`s to built-in arrays because they're safer, and they do not become pointers when you pass them to functions.[10] C++20's new `std::to_array` function[11] (header `<array>`) makes it convenient to create a `std::array` from a built-in array or an initializer list. Figure 7.6 demonstrates `to_array`. We use a generic lambda expression (lines 9–15) to display each `std::array`'s contents. Again, specifying a lambda parameter's type as `auto` enables the compiler to infer the parameter's type, based on the context in which the lambda appears. In this program, the generic lambda automatically determines the element type of the `std::array` over which it iterates.

10. C++ Core Guidelines, "SL.con.1: Prefer Using STL array or vector Instead of a C array." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-arrays.

11. Zhihao Yuan, "to_array from LFTS with Updates," July 17, 2019. Accessed December 31, 2021. https://wg21.link/p0325.

```
1   // fig07_06.cpp
2   // C++20: Creating std::arrays with to_array.          20
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <array>
6
7   int main() {
8      // generic lambda to display a collection of items
9      const auto display{
```

```
10          [](const auto& items) {
11             for (const auto& item : items) {
12                std::cout << fmt::format("{} ", item);
13             }
14          }
15       };
16
17       const int values1[]{10, 20, 30};
18
19       // creating a std::array from a built-in array
20       const auto array1{std::to_array(values1)};
21
22       std::cout <<fmt::format("array1.size() = {}\n", array1.size())
23          <<"array1: ";
24       display(array1); // use lambda to display contents
25
26      // creating a std::array from an initializer list
27      const auto array2{std::to_array({1, 2, 3, 4})};
28      std::cout << fmt::format("\n\narray2.size() = {}\n", array2.size())
29         << "array2: ";
30      display(array2); // use lambda to display contents
31
32      std::cout << '\n';
33   }
```

```
array1.size() = 3
array1: 10 20 30

array2.size() = 4
array2: 1 2 3 4
```

Fig. 7.6 C++20: Creating `std::arrays` with `to_array`.

Using `to_array` to Create a `std::array` from a Built-In Array

Line 20 creates a three-element `std::array` of `int`s by copying the contents of built-in array `values1`. We use `auto` to infer the `std::array` variable's type and size. If we declare the array's type and size explicitly and it does not match `to_array`'s return value, a compilation error occurs. We assign the result to the variable `array1`. Lines 22 and 24 display the `std::array`'s size and contents to confirm that it was created correctly.

Using `to_array` to create a `std::array` from an Initializer List

Line 27 shows that `to_array` can create a `std::array` from an initializer list. Lines 28 and 30 display the `array`'s size and contents to confirm that it was created correctly.

7.7 Using `const` with Pointers and the Data Pointed To

This section discusses how to combine `const` with pointer declarations to enforce the principle of least privilege. Chapter 5 explained that pass-by-value copies an argument's value into a function's parameter. If the copy is modified in the called function, the original value in the caller does not change. In some instances, even the copy of the argument's value should not be altered in the called function.

If a value does not (or should not) change in the body of a function to which it's passed, declare the parameter `const`. Before using a function, check its function prototype to determine the parameters it can and cannot modify.

There are four ways to declare pointers:

- a nonconstant pointer to nonconstant data,
- a nonconstant pointer to constant data (Fig. 7.7),
- a constant pointer to nonconstant data (Fig. 7.8) and
- a constant pointer to constant data (Fig. 7.9).

Each combination provides a different level of access privilege.

7.7.1 Using a Nonconstant Pointer to Nonconstant Data

The highest privileges are granted by a **nonconstant pointer to nonconstant data**:

- the data can be modified through the dereferenced pointer, and
- the pointer can be modified to point to other data.

Such a pointer's declaration (e.g., `int* countPtr`) does not include `const`.

7.7.2 Using a Nonconstant Pointer to Constant Data

A **nonconstant pointer to constant data** is

- a pointer that can be modified to point to any data of the appropriate type, but
- the data to which it points cannot be modified through that pointer.

The declaration for such a pointer places `const` to the left of the pointer's type, as in [12]

```
const int* countPtr;
```

12. Some programmers prefer to write this as int const* countPtr;. They'd read this declaration from right to left as "countPtr is a pointer to a constant integer."

The declaration is read from right to left as "`countPtr` is a pointer to an integer constant" or, more precisely, "`countPtr` is a nonconstant pointer to an integer constant."

Figure 7.7 demonstrates the GNU C++ compilation error produced when you try to modify data via a nonconstant pointer to constant data.

```
1  // fig07_07.cpp
2  // Attempting to modify data through a
3  // nonconstant pointer to constant data.
4
5  int main() {
6     int y{0};
7     const int* yPtr{&y};
8     *yPtr = 100; // error: cannot modify a const object
9  }
```

GNU C++ compiler error message:

```
fig07_07.cpp: In function 'int main()':
fig07_07.cpp:8:10: error: assignment of read-only location '* yPtr'
    8 |    *yPtr = 100; // error: cannot modify a const object
      |    ~~~~~~^~~~~
```

Fig. 7.7 Attempting to modify data through a nonconstant pointer to `const` data.

Perf Sec Use pass-by-value to pass fundamental-type arguments (e.g., `int`s, `double`s, etc.) unless the called function must directly modify the value in the caller. This is another example of the principle of least privilege. If large objects do not need to be modified by a called function, pass them using references or pointers to constant data—though references are preferred. This gives the performance benefits of pass-by-reference and avoids the copy overhead of pass-by-value. Passing large objects using references to constant data or pointers to constant data also offers the security of pass-by-value.

7.7.3 Using a Constant Pointer to Nonconstant Data

A **constant pointer to nonconstant data** is a pointer that

- always points to the same memory location and
- the data at that location can be modified through the pointer.

Pointers that are declared `const` must be initialized when they're declared. If the pointer is a function parameter, it's initialized with the pointer that's passed to the function. Each successive call to the function reinitializes that function parameter.

Figure 7.8 attempts to modify a constant pointer. Line 9 declares pointer `ptr` to be of type `int* const`. The declaration is read from right to left as "`ptr` is a constant pointer to a nonconstant integer." The pointer is initialized with the address of integer variable `x`. Line 12 attempts to assign the address of `y` to `ptr`, but the compiler generates an error message. No error occurs when line 11 assigns the value `7` to `*ptr`. The nonconstant value to which `ptr` points can be modified using the dereferenced `ptr`, even though `ptr` itself has been declared `const`.

```
1  // fig07_08.cpp
2  // Attempting to modify a constant pointer to nonconstant data.
3
4  int main() {
5     int x, y;
6
7     // ptr is a constant pointer to an integer that can be modified
8     // through ptr, but ptr always points to the same memory location.
9     int* const ptr{&x}; // const pointer must be initialized
10
11    *ptr = 7; // allowed: *ptr is not const
12    ptr = &y; // error: ptr is const; cannot assign to it a new address
13 }
```

Microsoft Visual C++ compiler error message:

```
error C3892: 'ptr': you cannot assign to a variable that is const
```

Fig. 7.8 Attempting to modify a constant pointer to nonconstant data.

7.7.4 Using a Constant Pointer to Constant Data

The minimum access privileges are granted by a **constant pointer to constant data**:

- such a pointer always points to the same memory location and
- the data at that location cannot be modified via the pointer.

Figure 7.9 declares pointer variable `ptr` to be of type `const int* const` (line 12). This declaration is read from right to left as "`ptr` is a constant pointer to an integer constant." The figure shows the error messages from Clang in Xcode generated by attempting to modify the data to which `ptr` points (line 16) and attempting to modify the address stored in the pointer variable (line 17). In line 14, no errors occur because *neither* the pointer *nor* the data it points to is being modified.

```
1   // fig07_09.cpp
2   // Attempting to modify a constant pointer to constant data.
3   #include <iostream>
4
5   int main() {
6      int x{5};
7      int y{6};
8
9      // ptr is a constant pointer to a constant integer.
10     // ptr always points to the same location; the integer
11     // at that location cannot be modified.
12     const int* const ptr{&x};
13
14     std::cout << *ptr << << *ptr << '\n';
15
16     *ptr = 7; // error: *ptr is const; cannot assign new value
17     ptr = &y; // error: ptr is const; cannot assign new address
18  }
```

Compiler error messages from Clang in Xcode:

```
fig07_09.cpp:16:9: error: read-only variable is not assignable
   *ptr = 7; // error: *ptr is const; cannot assign new value
   ~~~~ ^
fig07_09.cpp:17:8: error: cannot assign to variable 'ptr' with const-quali-
fied type 'const int *const'
   ptr = &y; // error: ptr is const; cannot assign new address
   ~~~ ^
```

Fig. 7.9 Attempting to modify a constant pointer to constant data.

7.8 sizeof Operator

The compile-time unary operator `sizeof` determines the size in bytes of a built-in array, type, variable or constant *during program compilation*. When applied to a built-in array's name, as in Fig. 7.10[13] (line 13), `sizeof` returns the total number of bytes in the built-in array as a value of type `size_t`. The computer we used to compile this program stores `double` variables in 8 bytes of memory. Array `numbers` is declared to have 20 elements (line 10), so it uses 160 bytes in memory. Applying `sizeof` to a pointer (line 21) returns the size of the pointer in bytes (4 on the system we used). 13. This is a mechanical example to demonstrate how sizeof works. If you use static code-analysis tools, such as the C++ Core Guidelines checker in Microsoft Visual Studio, you'll receive warnings because you should not pass built-in arrays to functions.

```
1   // fig07_10.cpp
```

```
   // Sizeof operator when used on a built-in array's name
   // returns the number of bytes in the built-in array.
   #include <fmt/format.h>
   #include <iostream>

   size_t getSize(double* ptr); // prototype

   int main() {
      double numbers[20]; // 20 doubles; occupies 160 bytes on our system

      std::cout << fmt::format("Number of bytes in numbers is {}\n\n",
         sizeof(numbers));

      std::cout << fmt::format("Number of bytes returned by getSize is {}\n",
         getSize(numbers));
   }

   // return size of ptr
   size_t getSize(double* ptr) {
      return sizeof(ptr);
   }
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Fig. 7.10 `sizeof` operator when applied to a built-in array's name returns the number of bytes in the built-in array.

Determining the Sizes of the Fundamental Types, a Built-In Array and a Pointer

11 Figure 7.11 uses `sizeof` to calculate the number of bytes used to store various standard data types. The output was produced using the Clang compiler in Xcode. Type sizes are platform-dependent. When we run this program on our Windows system, `long` is 4 bytes and `long long` is 8 bytes, whereas they're both 8 bytes on our Mac. In this example, lines 7–15 implicitly initialize each variable to 0 using a C++11 empty initializer list, `{}`.[14]

14. Line 16 uses const rather than constexpr to prevent a type mismatch compilation error. The name of the built-in array of ints (line 15) decays to a const int*, so we must declare ptr with that type.

```
   // fig07_11.cpp
   // sizeof operator used to determine standard data type sizes.
   #include <fmt/format.h>
   #include <iostream>

   int main() {
      constexpr char c{}; // variable of type char
      constexpr short s{}; // variable of type short
      constexpr int i{}; // variable of type int
      constexpr long l{}; // variable of type long
      constexpr long long ll{}; // variable of type long long
      constexpr float f{}; // variable of type float
      constexpr double d{}; // variable of type double
      constexpr long double ld{}; // variable of type long double
      constexpr int array[20]{}; // built-in array of int
      constexpr int* const ptr{array}; // variable of type int*

```

```
18       std::cout << fmt::format("sizeof c = {}\tsizeof(char) = {}\n",
19          sizeof c, sizeof(char));
20       std::cout << fmt::format("sizeof s = {}\tsizeof(short) = {}\n",
21          sizeof s, sizeof(short));
22       std::cout << fmt::format("sizeof i = {}\tsizeof(int) = {}\n",
23          sizeof i, sizeof(int));
24       std::cout << fmt::format("sizeof l = {}\tsizeof(long) = {}\n",
25          sizeof l, sizeof(long));
26       std::cout << fmt::format("sizeof ll = {}\tsizeof(long long) = {}\n",
27          sizeof ll, sizeof(long long));
28       std::cout << fmt::format("sizeof f = {}\tsizeof(float) = {}\n",
29          sizeof f, sizeof(float));
30       std::cout << fmt::format("sizeof d = {}\tsizeof(double) = {}\n",
31          sizeof d, sizeof(double));
32       std::cout << fmt::format(sizeof ld = {}\tsizeof(long double) = {}\n",
33          sizeof ld, sizeof(long double));
34       std::cout << fmt::format("sizeof array = {}\n",  sizeof array);
35       std::cout << fmt::format("sizeof ptr = {}\n",  sizeof ptr);
36    }
```

```
sizeof c = 1     sizeof(char) = 1
sizeof s = 2     sizeof(short) = 2
sizeof i = 4     sizeof(int) = 4
sizeof l = 8     sizeof(long) = 8
sizeof ll = 8    sizeof(long long) = 8
sizeof f = 4     sizeof(float) = 4
sizeof d = 8     sizeof(double) = 8
sizeof ld = 16   sizeof(long double) = 16
sizeof array = 80
sizeof ptr = 8
```

Fig. 7.11 `sizeof` operator used to determine standard data type sizes.

11 The number of bytes used to store a particular data type may vary among systems and compilers. When writing programs that depend on data type sizes, consider using the **fixed-size integer types** added in C++11 (header `<cstdint>`). For the complete list, see:

```
https://en.cppreference.com/w/cpp/types/integer
```

Operator `sizeof` can be applied to any expression or type name. When applied to a variable name or expression, the number of bytes used to store the corresponding type is returned. The parentheses used with `sizeof` are required only if a type name (e.g., `int`) is supplied as its operand. The parentheses used with `sizeof` are not required when `sizeof`'s operand is an expression. Remember that `sizeof` is a compile-time operator, so its operand will not be evaluated at runtime.

7.9 Pointer Expressions and Pointer Arithmetic

C++ enables **pointer arithmetic**—arithmetic operations that may be performed on pointers. This section describes the operators with pointer operands and how these operators are used with pointers.

CG Err Pointer arithmetic is appropriate only for pointers that point to built-in array elements. You're likely to encounter pointer arithmetic in legacy code. However, **the C++ Core**

Guidelines indicate that a pointer should refer only to a single object (not an array)[15] and that you should not use pointer arithmetic because it's highly error-prone.[16] If you need to process built-in arrays, use C++20 `span`s instead (Section 7.10).

15. C++ Core Guidelines, "ES.42: Keep Use of Pointers Simple and Straightforward." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr

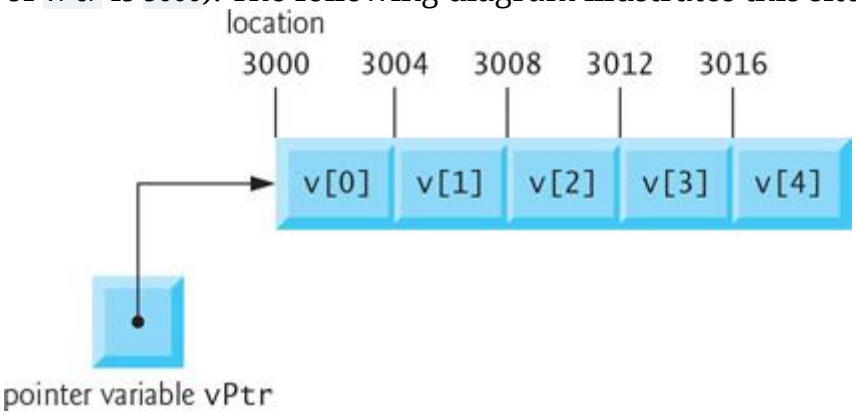
16. C++ Core Guidelines, "Pro.bounds: Bounds Safety Profile." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#SS-bounds.

Valid pointer arithmetic operations are

- incrementing (`++`) or decrementing (`--`),
- adding an integer to a pointer (`+` or `+=`) or subtracting an integer from a pointer (`-` or `-=`) and
- subtracting one pointer from another of the same type.

Pointer arithmetic results depend on the size of the memory objects a pointer points to, so pointer arithmetic is machine-dependent.

Most computers today have four-byte (32-bit) or eight-byte (64-bit) integers, though some of the billions of resource-constrained Internet of Things (IoT) devices are built using eight-bit or 16-bit hardware with two-byte integers—the minimum size for an `int` according to the C++ standard. Assume that `int v[5]` exists, that its first element is at memory location `3000` and that `int`s are stored in four bytes. Also, assume that pointer `vPtr` has been initialized to point to `v[0]` (i.e., the value of `vPtr` is `3000`). The following diagram illustrates this situation for a machine with four-byte integers:



Variable `vPtr` can be initialized to point to `v` with either of the following statements (because a built-in array's name implicitly converts to the address of its zeroth element):

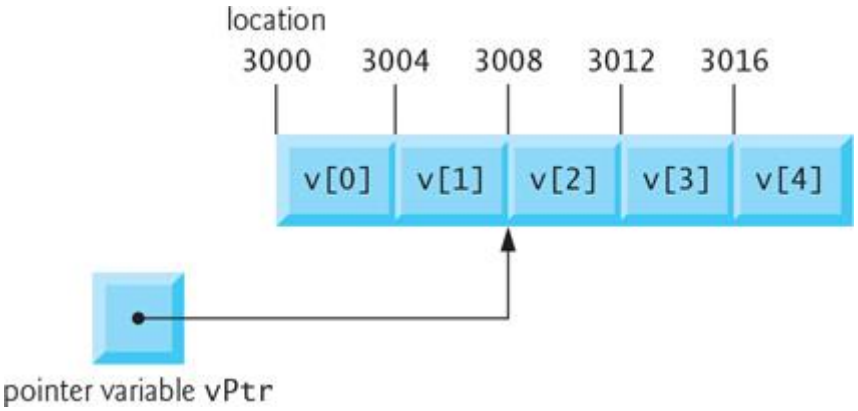
```
int* vPtr{v};
int* vPtr{&v[0]};
```

7.9.1 Adding Integers to and Subtracting Integers from Pointers

In conventional arithmetic, the addition `3000 + 2` yields the value `3002`. This is normally not the case with pointer arithmetic. Adding an integer to or subtracting an integer from a pointer increments or decrements the pointer by that integer times the size of the type to which the pointer refers. The number of bytes depends on the memory object's data type. For example, the statement

```
vPtr += 2;
```

would produce `3008` (`3000 + 2 * 4`), assuming that an `int` is stored in four bytes of memory. In the built-in array `v`, `vPtr` would now point to `v[2]` as in the diagram below:



If `vPtr` had been incremented to `3016`, which points to `v[4]`, the statement

```
vPtr -= 4;
```

would set `vPtr` back to `3000`—the beginning of the built-in array. If a pointer is being incremented or decremented by one, the increment (`++`) and decrement (`--`) operators can be used. Each of the statements

```
++vPtr;
vPtr++;
```

increments the pointer to point to the built-in array's next element. Each of the statements

```
--vPtr;
vPtr--;
```

decrements the pointer to point to the built-in array's previous element.

CG **There's no bounds checking on pointer arithmetic, so the C++ Core Guidelines recommend using `std::span`s instead, which we demonstrate in Section 7.10.** You must ensure that every pointer arithmetic operation that adds an integer to or subtracts an integer from a pointer results in a pointer that references an element within the built-in array's bounds. As you'll see, **`std::span`s have bounds checking, which helps you avoid errors.**

7.9.2 Subtracting One Pointer from Another

Pointer variables pointing to the same built-in array may be subtracted from one another. For example, if `vPtr` contains the address `3000` and `v2Ptr` contains the address `3008`, the statement

```
x = v2Ptr - vPtr;
```

would assign to `x` the number of built-in array elements from `vPtr` to `v2Ptr`—in this case, 2. We cannot assume that two variables of the same type are stored contiguously in memory unless they're adjacent elements of a built-in array. Subtracting or comparing two pointers that do not refer to elements of the same built-in array is a logic error.

7.9.3 Pointer Assignment

A pointer can be assigned to another pointer if both pointers are of the same type.[17] Also, any pointer to a fundamental type or class type can be assigned to a **`void*`** (**pointer to `void`**) without casting.

A `void*` pointer can represent any pointer type. However, a pointer of type `void*` cannot be assigned directly to a pointer of another type—the pointer of type `void*` must first be `static_cast` to the proper pointer type.
17. Of course, `const` pointers cannot be modified.

7.9.4 Cannot Dereference a `void*`

A `void*` pointer cannot be dereferenced. For example, the compiler "knows" that an `int*` points to four bytes of memory on a machine with four-byte integers. Dereferencing an `int*` creates an *lvalue* that is an alias for the `int`'s four bytes in memory. A `void*`, however, simply contains a memory address for an unknown data type. You cannot use a `void*` in pointer arithmetic, nor can you dereference a `void*` because the compiler does not know the type or size of the data to which the pointer refers.

The allowed operations on `void*` pointers are:

- comparing `void*` pointers with other pointers,
- casting `void*` pointers to other pointer types and
- converting other pointer types to `void*` pointers.

All other operations on `void*` pointers are compilation errors.

7.9.5 Comparing Pointers

Pointers can be compared using equality and relational operators. Relational comparisons (`<`, `<=`, `>` and `>=`) are meaningless unless the pointers point to elements of the same built-in array. Pointer comparisons compare the addresses stored in the pointers. Comparing two pointers pointing to the same built-in array could show, for example, that one pointer points to a higher-numbered element than the other. A common use of pointer equality comparison is determining whether a pointer has the value `nullptr` (i.e., a pointer to nothing).

20 7.10 Objects-Natural Case Study: C++20 `span`s—Views of Contiguous Container Elements

We now continue our Objects-Natural approach by taking C++20 `span` objects for a spin.
A `span` (header `<span>`) enables programs to view contiguous elements of a container, such as a built-in array, a `std::array` or a `std::vector`. A `span` is a "view" into a container. It "sees" the container's elements but does not have its own copy of those elements.

CG Earlier, we discussed how C++ built-in arrays decay to pointers when passed to functions. In particular, the function's parameter loses the size information provided when you declared the array. You saw this in our `sizeof` demonstration in Fig. 7.10. **The C++ Core Guidelines recommend passing built-in arrays to functions as `span`s[18], which represent both a pointer to the array's first element and the array's size.** Figure 7.12 demonstrates some key `span` capabilities. We broke this program into parts for discussion purposes.
18. C++ Core Guidelines, "R.14: Avoid [] Parameters, Prefer span." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-ap.

```
1  // fig07_12.cpp
2  // C++20 spans: Creating views into containers.
3  #include <array>
4  #include <fmt/format.h>
```

```
5   #include <iostream>
6   #include <numeric>
7   #include <span>
8   #include <vector>
```

Fig. 7.12 C++20 spans: Creating views into containers.

Function displayArray

Sec CG Passing a built-in array to a function typically requires both the array's name and the array's size. Though the parameter items (line 12) is declared with [], it's simply a pointer to an int. The pointer does not "know" how many elements the function's argument contains. There are various problems with this approach. For instance, the code that calls displayArray could pass the wrong value for size. In this case, the function might not process all of items' elements, or the function might access an element outside items' bounds—a logic error and a potential security issue. In addition, we previously discussed the disadvantages of external iteration, as used in lines 13–15. The C++ Core Guidelines checker in Visual Studio issues several warnings about displayArray and passing built-in arrays to functions. We include function displayArray in this example only to compare it with passing spans in function displaySpan, which is the recommended approach.

```
10  // items parameter is treated as a const int* so we also need the size to
11  // know how to iterate over items with counter-controlled iteration
12  void displayArray(const int items[], size_t size) {
13     for (size_t i{0}; i < size; ++i) {
14        std::cout << fmt::format("{} ", items[i]);
15     }
16  }
```

Function displaySpan

CG The C++ Core Guidelines indicate that a pointer should point only to one object, not an array,[19] and that functions like displayArray, which receive a pointer and a size, are error-prone.[20] To fix these issues, you should pass arrays to functions using spans. Function displaySpan (lines 20–24) receives a span containing const ints—const because the function does not need to modify the data.

19. C++ Core Guidelines, "ES.42: Keep Use of Pointers Simple and Straightforward." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-ptr.

20. C++ Core Guidelines, "I.13: Do Not Pass an Array as a Single Pointer." Accessed January 2, 2022.0 https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-array.

```
18  // span parameter contains both the location of the first item
19  // and the number of elements, so we can iterate using range-based for
20  void displaySpan(std::span<const int> items) {
21     for (const auto& item : items) { // spans are iterable
22        std::cout << fmt::format("{} ", item);
23     }
24  }
```

CG Perf A `span` encapsulates both a pointer and a `size_t` representing the number of elements. When you pass a built-in array (or a `std::array` or `std::vector`) to `displaySpan`, C++ implicitly creates a `span` containing a pointer to the array's first element and its size, which the compiler determines from the array's declaration. This `span` views the data in the original array that you pass as an argument. The C++ Core Guidelines indicate that you can pass a `span` by value because it's just as efficient as passing the pointer and size separately,[21] as we did in `displayArray`.

21. C++ Core Guidelines, "F.24: Use a span<T> or a span_p<T> to Designate a Half-open Sequence." Accessed January 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-range.

Sec A `span` has many capabilities similar to `array`s and `vector`s, such as iteration via the range-based `for` statement. Because a `span` is created based on the array's original size as determined by the compiler, the range-based `for` guarantees that we cannot access an element outside the array's bounds, thus fixing the various problems associated with `displayArray` and helping prevent security issues like buffer overflows.

Function `times2`

Because a `span` is a view into an existing container, changing the `span`'s elements changes the container's original data. Function `times2` multiplies every item in its `span<int>` by 2. Note that we use a non-`const` reference to modify each element that the `span` views.

```
26  // spans can be used to modify elements in the original data structure
27  void times2(std::span<int> items) {
28     for (int& item : items) {
29        item *= 2;
30     }
31  }
```

Passing an Array to a Function to Display the Contents

Lines 34–36 create the `int` built-in array `values1`, the `std::array values2` and the `std::vector values3`. Each has five elements and stores its elements contiguously in memory. Line 41 calls `displayArray` to display `values1`'s contents. The `displayArray` function's first parameter is a pointer to an `int`, so we cannot use a `std::array`'s or `std::vector`'s name to pass these objects to `displayArray`.

```
33 int main() {
34    int values1[]{1, 2, 3, 4, 5};
35    std::array values2{6, 7, 8, 9, 10};
36    std::vector values3{11, 12, 13, 14, 15};
37
38    // must specify size because the compiler treats displayArray's items
39    // parameter as a pointer to the first element of the argument
40    std::cout << "values1 via displayArray: ";
41    displayArray(values1, 5);
```

```
values1 via displayArray: 1 2 3 4 5
```

Implicitly Creating spans and Passing Them to Functions

Line 46 calls `displaySpan` with `values1` as an argument. The function's parameter was declared as

```
std::span<const int>
```

so C++ creates a `span` containing a `const int*` that points to the array's first element and a `size_t` representing the array's size, which the compiler gets from the `values1` declaration (line 34). Because `span`s can view any contiguous sequence of elements, you may also pass a `std::array` or `std::vector` of `int`s to `displaySpan` (lines 50 and 52). C++ will create an appropriate `span` representing a pointer to the container's first element and size. This makes function `displaySpan` more flexible than `displayArray`, which could receive only the built-in array in this example.

```
43    // compiler knows values1's size and automatically creates a span
44    // representing &values1[0] and the array's length
45    std::cout << "\nvalues1 via displaySpan: ";
46    displaySpan(values1);
47
48    // compiler also can create spans from std::arrays and std::vectors
49    std::cout << "\nvalues2 via displaySpan: ";
50    displaySpan(values2);
51    std::cout << "\nvalues3 via displaySpan: ";
52    displaySpan(values3);
53
values1 via displaySpan: 1 2 3 4 5
values2 via displaySpan: 6 7 8 9 10
values3 via displaySpan: 11 12 13 14 15
```

Changing a `span`'s Elements Modifies the Original Data

As we mentioned, function `times2` multiplies its `span`'s elements by 2. Line 55 calls `times2` with `values1` as an argument. The function's parameter was declared as

```
std::span<int>
```

so C++ creates a `span` containing an `int*` that points to the array's first element and a `size_t` representing the array's size, which the compiler gets from the `values1` declaration (line 34). To prove that `times2` modified the original array's data, line 57 displays `values1`'s updated values. Like `displaySpan`, `times2` can be called with this program's `std::array` or `std::vector` as well.

```
54    // changing a span's contents modifies the original data
55    times2(values1);
56    std::cout << "\n\nvalues1 after times2 modifies its span argument: ";
57    displaySpan(values1);
values1 after times2 modifies its span argument: 2 4 6 8 10
```

Manually Creating a Span and Interacting with It

You can explicitly create `span`s and interact with them. Line 60 creates a `span<int>` that views the data in `values1`—the compiler uses CTAD to infer the element type `int` from `values1`'s elements. Lines 61–62 demonstrate the span's `front` and `back` member functions, which return the first and last elements of the view—thus, the first and last elements of `values1`, respectively.

```
59     // spans have various array-and-vector-like capabilities
60     std::span mySpan{values1}; // span<int>
61     std::cout << "\n\nmySpan's first element: " << mySpan.front()
62        << "\nmySpan's last element: " << mySpan.back();
```

```
mySpan's first element: 2
mySpan's last element: 10
```

CG **A philosophy of the C++ Core Guidelines is to "prefer compile-time checking to runtime checking."**[22] This enables the compiler to find and report errors at compile-time, rather than you writing code to help prevent runtime errors. In line 60, the compiler determines the `span`'s size (`5`) from the `values1` declaration in line 34. You can explicitly state the `span`'s type and size, as in

22. C++ Core Guidelines, "P.5: Prefer Compile-Time Checking to Run-Time Checking." Accessed January 31, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rp-compile-time.

```
span<int, 5> mySpan{values1};
```

In this case, the compiler ensures that the `span`'s declared size matches `values1`'s size; otherwise, a compilation error occurs.

Using a `span` with the Standard Library's `accumulate` Algorithm

As you've seen in this example, `span`s are iterable. This means you also can use the `begin` and `end` functions with `span`s to pass them to C++ standard library algorithms, such as `accumulate` (line 66) or `sort`. We cover standard library algorithms in depth in Chapter 14.

```
64     // spans can be used with standard library algorithms
65     std::cout << "\n\nSum of mySpan's elements: "
66        << std::accumulate(std::begin(mySpan), std::end(mySpan), 0);
```

```
Sum of mySpan's elements: 30
```

Creating Subviews

Sometimes, you might want to process subsets of a `span`. A `span`'s `first`, `last` and `sub-span` member functions create subviews. Lines 70 and 72 use `first` and `last` to get `span`s representing `values1`'s first three and last three elements, respectively. Line 74 uses `sub-span` to get a `span` that views the `3` elements starting from index `1`. In each case, we pass the subview to `displaySpan` to confirm what the subview represents.

```
68     // spans can be used to create subviews of a container
69     std::cout << "\n\nFirst three elements of mySpan: ";
70     displaySpan(mySpan.first(3));
```

```
71     std::cout << "\nLast three elements of mySpan: ";
72     displaySpan(mySpan.last(3));
73     std::cout << "\nMiddle three elements of mySpan: ";
74     displaySpan(mySpan.subspan(1, 3));
```

```
First three elements of mySpan: 2 4 6
Last three elements of mySpan: 6 8 10
Middle three elements of mySpan: 4 6 8
```

Changing a Subview's Elements Modifies the Original Data

A subview of non-`const` data can modify that data. Line 77 passes to function `times2` a `span` that views the `3` elements starting from index `1` of `values1`. Line 79 displays the updated `values1` elements to confirm the results.

```
76     // changing a subview's contents modifies the original data
77     times2(mySpan.subspan(1, 3));
78     std::cout << "\n\nvalues1 after modifying elements via span: ";
79     displaySpan(values1);
```

```
values1 after modifying elements via span: 2 8 12 16 10
```

Accessing a View's Elements Via the `[]` Operator

Like built-in arrays, `std::array`s and `std::vector`s, you can access and modify `span` elements via the `[]` operator, which does not provide range checking. Line 82 displays the element at index `2`.[23] [23].`std::span` does not provide a range-checked `at` member function, though it might in the future.

```
81     // access a span element via []
82     std::cout << "\n\nThe element at index 2 is: " << mySpan[2];
83  }
```

```
The element at index 2 is: 12
```

7.11 A Brief Intro to Pointer-Based Strings

We've already used the C++ standard library `string` class to represent strings as full-fledged objects. Chapter 8 presents class `std::string` in detail. This section introduces pointer-based strings, as inherited from the C programming language. Here, we'll refer to these as **C-strings** or strings and use `std::string` when referring to the C++ standard library's `string` class.

Sec `std::string` is preferred because it eliminates many of the security problems and bugs caused by manipulating C-strings. However, there are some cases in which C-strings are required, such as when processing command-line arguments. Also, if you work with legacy C and C++ programs, you're likely to encounter pointer-based strings. We cover C-strings in detail in online Appendix E.

Characters and Character Constants

Characters are the fundamental building blocks of C++ source programs. Every program is composed of characters that—when grouped meaningfully—are interpreted by the compiler as instructions and

data used to accomplish a task. A program may contain **character constants**, each of which is an integer value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set. For example, `'z'` represents the letter `z`'s integer value (122 in the ASCII character set; Appendix B) as type `char`. Similarly, `'\n'` represents the integer value of newline (10 in the ASCII character set).

Pointer-Based Strings

A C-string is a built-in array of characters ending with a **null character (`'\0'`)**, which marks where the string terminates in memory. A C-string is accessed via a pointer to its first character (no matter how long the string is).

String Literals as Initializers

A string literal may be used as an initializer for a built-in array of `char`s or a variable of type `const char*`. The declarations

```
char color[]{"blue"};
const char* colorPtr{"blue"};
```

each initialize a variable to the string `"blue"`. The first declaration creates a five-element built-in array `color` containing the characters `'b'`, `'l'`, `'u'`, `'e'` and `'\0'`. The second creates the pointer variable `colorPtr` pointing to the letter `b` in the string `"blue"` (which ends in `'\0'`) somewhere in memory. The first declaration above also may be implemented using an initializer list of individual characters in which you manually include the terminating `'\0'`, as in:

```
char color[]{'b', 'l', 'u', 'e', '\0'};
```

String literals exist for the duration of the program. They may be shared if the same string literal is referenced from multiple locations in a program. String literals are immutable—they cannot be modified.

Problems with C-Strings

Not allocating sufficient space in a built-in array of `char`s to store the null character that terminates a string is a logic error. Creating or using a C-string that does not contain a terminating null character can lead to logic errors.

Sec When storing a string of characters in a built-in array of `char`s, be sure that the builtin array is large enough to hold the largest string that will be stored. C++ allows strings of any length. If a string is longer than the built-in array of `char`s in which it's to be stored, characters beyond the end of the built-in array will overwrite subsequent memory locations. This could lead to logic errors, program crashes or security breaches.

Displaying C-Strings

A built-in array of `char`s representing a null-terminated string can be output with `cout` and `<<`. The statement

```
std::cout << color;
```

displays the built-in array `color`. The `cout` object does not care how large the built-in array of `char`s is. The characters are output until a terminating null character is encountered. The null character is not

displayed. Both `cin` and `cout` assume that built-in arrays of `char`s should be processed as strings terminated by null characters. They do not provide similar input and output processing capabilities for other built-in array types.

7.11.1 Command-Line Arguments

There are cases in which built-in arrays and C-strings must be used. For example, **command-line arguments** are often passed to applications to specify configuration options, file names to process and more. You supply command-line arguments to a program by placing them after its name when executing it from the command line. On a Windows system, the command

```
dir /p
```

uses the `/p` argument to list the contents of the current directory, pausing after each screen of information. Similarly, on Linux or macOS, the following command uses the `-la` argument to list the contents of the current directory with details about each file and directory:

```
ls -la
```

Command-line arguments are passed into a C++ program as C-strings. The application name is treated as the first command-line argument. To use the arguments as `std::string`s or other data types (`int`, `double`, etc.), you must convert the arguments to those types. Figure 7.13 displays the number of command-line arguments passed to the program, then displays each argument on a separate line.

```
1   // fig07_13.cpp
2   // Reading in command-line arguments.
3   #include <fmt/format.h>
4   #include <iostream>
5
6   int main(int argc, char* argv[]) {
7      std::cout << fmt::format("Number of arguments: {}\n\n", argc);
8
9      for (int i{0}; i < argc; ++i) {
10        std::cout << fmt::format("{}\n", argv[i]);
11     }
12  }
```

```
fig07_13 Amanda Green 97
Number of arguments: 4

fig07_13
Amanda
Green
97
```

Fig. 7.13 Reading in command-line arguments.

To receive command-line arguments, declare `main` with two parameters (line 6), which by convention are named `argc` and `argv`, respectively. The first is an `int` representing the number of arguments. The second is a pointer to the first element of a built-in array of `char*`—some

programmers write this as `char** argv`. The first element of the array is a C-string for the application name.[24] The remaining elements are C-strings for the other command-line arguments.

24. The C++ standard allows implementations to return an empty string for `argv[0]`.

The command

```
fig07_13 Amanda Green 97
```

passes `"Amanda"`, `"Green"` and `97"` to the application `fig07_13`. On macOS and Linux, you'd run this program with `"./fig07_13"`. Command-line arguments are separated by whitespace, *not* commas. When this command executes, `fig07_13`'s `main` function receives the argument count `4` and a four-element array of C-strings:

- `argv[0]` contains the application's name `"fig07_13"` (or `"./fig07_13"` on macOS or Linux) and
- `argv[1]` through `argv[3]` contain `"Amanda"`, `"Green"` and `"97"`, respectively.

You determine how to use these arguments in your program. Due to the problems we've discussed with C-strings, you should convert the command-line arguments to `std::string`s before using them in your program.

20 7.11.2 Revisiting C++20's `to_array` Function

Section 7.6 demonstrated converting built-in arrays to `std::array`s with `to_array`. Figure 7.14 shows another purpose of `to_array`. We use the same lambda expression (lines 9–13) as in Fig. 7.6 to display the `std::array` contents after the `to_array` call.

```
1  // fig07_14.cpp
2  // C++20: Creating std::arrays from string literals with to_array.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <array>
6
7  int main() {
8     // lambda to display a collection of items
9     const auto display{
10       [](const auto& items) {
11          for (const auto& item : items) {
12             std::cout << fmt::format("{} ", item);
13          }
14       }
15    };
16
17    // initializing an array with a string literal
18    // creates a one-element array<const char*>
19    const auto array1{std::array{"abc"}};
20    std::cout << fmt::format("array1.size() = {}\narray1: ",
21       array1.size());
22    display(array1); // use lambda to display contents
23
24    // creating std::array of characters from a string literal
25    const auto array2{std::to_array("C++20")};
26    std::cout << fmt::format("\n\narray2.size() = {}\narray2: ",
27       array2.size());
```

```
28      display(array2); // use lambda to display contents
29
30      std::cout << '\n';
31   }
array1.size() = 1
array1: abc

array2.size() = 6
array2: C + + 2 0
```

Fig. 7.14 C++20: Creating `std::array`s from string literals with `to_array`.

Initializing a `std::array` from a String Literal Creates a One-Element `array`

Line 19 creates a one-element `array` containing a `const char*` pointing to the C-string `"abc"`.

Passing a String Literal to `to_array` Creates a `std::array` of `char`

On the other hand, passing a string literal to `to_array` (line 25) creates a `std::array` of `char`s containing elements for each character and the terminating null character. Lines 25–26 confirm that the `array`'s size is `6`. Line 27 confirms the `array`'s contents. The null character does not have a visual representation, so it does not appear in the output.

7.12 Looking Ahead to Other Pointer Topics

In later chapters, we'll introduce additional pointer topics:

- In Chapter 10, OOP: Inheritance and Runtime Polymorphism, we'll use pointers with class objects to show that the "runtime polymorphic processing" associated with object-oriented programming can be performed with references or pointers—you should favor references.
- In Chapter 11, Operator Overloading, Copy/Move Semantics and Smart Pointers, we introduce dynamic memory management with pointers, which allows you at execution-time to create and destroy objects as needed. Improperly managing this process is a source of subtle errors, such as "memory leaks." We'll show how "smart pointers" can automatically manage memory and other resources that should be returned to the operating system when they're no longer needed.
- In Chapter 14, Standard Library Algorithms and C++20 Ranges & Views, we show that a function's name can be treated as a pointer to its implementation and that functions can be passed into other functions via function pointers.

7.13 Wrap-Up

This chapter discussed pointers, built-in pointer-based arrays and pointer-based strings (C-strings). **We pointed out Modern C++ guidelines that recommend avoiding most pointers—preferring references over pointers, `std::array` and `std::vector` objects to built-in arrays, and `std::string` objects to C-strings.**

We declared and initialized pointers and demonstrated the pointer operators `&` and `*`. We showed that pointers enable pass-by-reference, but you should generally prefer references for that purpose. We used built-in, pointer-based arrays and showed their intimate relationship with pointers.

We discussed various combinations of `const` with pointers and the data they point to and used the `sizeof` operator to determine the number of bytes that store values of particular fundamental types and pointers. We demonstrated pointer expressions and pointer arithmetic.

We briefly discussed C-strings then showed how to process command-line arguments—a simple task for which C++ still requires you to use both pointer-based C-strings and pointer-based arrays.

As a reminder, the key takeaway from reading this chapter is that you should avoid using pointers, pointer-based arrays and pointer-based strings whenever possible. For programs that still use pointer-based arrays, you can use C++20's `to_array` function to create `std::array`s from built-in arrays and C++20's `span`s as a safer way to process built-in pointer-based arrays. In the next chapter, we discuss typical string-manipulation operations provided by `std::string` and introduce file-processing capabilities.

8. `strings`, `string_views`, Text Files, CSV Files and Regex

Objectives

In this chapter, you'll:

- Determine `string` characteristics.
- Find, replace and insert characters in `strings`.
- Use C++11 numeric conversion functions.
- Use C++17 `string_views` for lightweight views of contiguous characters.
- Write and read sequential files.
- Perform input from and output to `strings` in memory.
- Do an Objects-Natural case study using an object of an open-source-library class to read and process data about the *Titanic* disaster from a CSV (comma-separated values) file.
- Do an Objects-Natural case study using C++11 regular expressions (regex) to search strings for patterns, validate data and replace substrings.

Outline

8.1 Introduction
8.2 `string` Assignment and Concatenation
8.3 Comparing `strings`
8.4 Substrings
8.5 Swapping `strings`
8.6 `string` Characteristics
8.7 Finding Substrings and Characters in a `string`
8.8 Replacing and Erasing Characters in a `string`
8.9 Inserting Characters into a `string`
8.10 C++11 Numeric Conversions
8.11 C++17 `string_view`
8.12 Files and Streams
8.13 Creating a Sequential File
8.14 Reading Data from a Sequential File
8.15 C++14 Reading and Writing Quoted Text
8.16 Updating Sequential Files
8.17 String Stream Processing
8.18 Raw String Literals
8.19 Objects-Natural Case Study: Reading and Analyzing a CSV File Containing *Titanic* Disaster Data
8.19.1 Using `rapidcsv` to Read the Contents of a CSV File
8.19.2 Reading and Analyzing the *Titanic* Disaster Dataset
8.20 Objects-Natural Case Study: Intro to Regular Expressions
8.20.1 Matching Complete Strings to Patterns
8.20.2 Replacing Substrings
8.20.3 Searching for Matches
8.21 Wrap-Up

8.1 Introduction

This chapter discusses additional `std::string` features and introduces `string_view`s, text file-processing, CSV file processing and regular expressions.

`std::string`s

We've been using `std::string` objects since Chapter 2. Here, we introduce many more `std::string` manipulations, including assignment, comparisons, extracting substrings, searching for substrings and modifying `std::string` objects. We also discuss converting `std::string` objects to numeric values and vice versa.

17 C++17 `string_view`s

We introduce C++17's `string_view`s—read-only views of C-strings or `std::string` objects. Like `std::span`, a `string_view` does not own the data it views. You'll see that `string_view`s have many similar capabilities to `std::string`s, making them appropriate for cases in which you do not need modifiable strings.

Text Files and String Stream Processing

Data storage in memory is temporary. **Files** are used for **data persistence**—permanent data retention. Computers store files on **secondary storage devices,** such as flash drives, and frequently today, in the cloud. We explain how to build C++ programs that create, update and process text files. We also show how to output data to and read data from a `std::string` in memory using `ostringstream`s and `istringstream`s.

Objects-Natural Case Study: CSV Files and the Titanic Disaster Dataset

This chapter's first of two Objects-Natural case studies introduces the CSV (comma-separated values) file format. CSV is popular for datasets used in big data, data analytics and data science, and artificial intelligence applications like natural language processing, machine learning and deep learning.

A commonly used dataset for data analytics and data science beginners is the Titanic disaster dataset. It lists all the passengers and whether they survived when the ship *Titanic* struck an iceberg and sank during its maiden voyage of April 10–15, 1912. We use a class from the open-source `rapidcsv` library to create an object that reads the Titanic dataset from a CSV file. Then, we view some of the data and perform some basic data analytics.

Objects-Natural Case Study: Using Regular Expressions to Search Strings for Patterns, Validate Data and Replace Substrings

11 This chapter's second Objects-Natural case study introduces regular expressions, which are particularly crucial in today's data-rich applications. We'll use C++11 `regex` objects to create regular expressions, then use them with various functions in the `<regex>` header to match patterns in text. Earlier chapters mentioned the importance of validating user input in industrial-strength code. The `std::string`, string stream and regular expression capabilities presented in this chapter are frequently used to validate data.

8.2 `string` Assignment and Concatenation

Figure 8.1 demonstrates various `std::string` assignment and concatenation capabilities.

```
1   // fig08_01.cpp
2   // Demonstrating string assignment and concatenation.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <string>
6
7   int main() {
8      std::string s1{"cat"};
9      std::string s2; // initialized to the empty string
10     std::string s3; // initialized to the empty string
11
12     s2 = s1; // assign s1 to s2
13     s3.assign(s1); // assign s1 to s3
14     std::cout << fmt::format("s1: {}\ns2: {}\ns3: {}\n\n", s1, s2, s3);
15
16     s2.at(0) = 'r'; // modify s2
17     s3.at(2) = 'r'; // modify s3
18     std::cout << fmt::format("After changes:\ns2: {}\ns3: {}", s2, s3);
19
20     std::cout << "\n\nAfter concatenations:\n";
21     std::string s4{s1 + "apult"}; // concatenation
22     s1.append("acomb"); // create "catacomb"
23     s3 += "pet"; // create "carpet" with overloaded +=
24     std::cout << fmt::format("s1: {}\ns3: {}\ns4: {}\n", s1, s3, s4);
25
26     // append locations 4 through end of s1 to
27     // create string "comb" (s5 was initially empty)
28     std::string s5; // initialized to the empty string
29     s5.append(s1, 4, s1.size() - 4);
30     std::cout << fmt::format("s5: {}", s5);
31  }
```

```
s1: cat
s2: cat
s3: cat

After changes:
s2: rat
s3: car

After concatenations:
s1: catacomb
s3: carpet
s4: catapult
s5: comb
```

Fig. 8.1 Demonstrating `string` assignment and concatenation.

String Assignment

Lines 8–10 create the `string`s `s1`, `s2` and `s3`. Line 12 uses the assignment operator to copy the contents of `s1` into `s2`. Line 13 uses member function `assign` to copy `s1`'s contents into `s3`. This particular version of `assign` is equivalent to using the `=` operator, but `assign` also has many overloads. For details of each, see

```
https://en.cppreference.com/w/cpp/string/basic_string/assign
```

For example, one overload copies a specified number of characters, as in

```
target.assign(source, start, numberOfChars);
```

where `source` is the `string` to copy, `start` is the starting index and `numberOfChars` is the number of characters to copy.

Accessing String Elements By Index

Lines 16–17 use the `string` member function **`at`** to assign `'r'` to `s2` at index `0` (forming `"rat"`) and to assign `'r'` to `s3` at index `2` (forming `"car"`). You also can use the member function `at` to get the character at a specific index in a `string`. As with `std::array` and `std::vector`, a `std::string`'s `at` member function performs range checking and throws an `out_of_range` exception if the index is not within the `string`'s bounds. The `string` subscript operator, `[]`, does not check whether the index is in bounds. This is consistent with its use with `std::array` and `std::vector`. You also can iterate through the characters in a string using range-based `for` as in

```
for (char c : s3) {
     cout << c;
}
```

which ensures that you do not access elements outside the `string`'s bounds.

Accessing String Elements By Index

Line 21 initializes `s4` to the contents of `s1`, followed by `"apult"`. For `std::string`, the `+` operator denotes string concatenation. Line 22 uses member function **`append`** to concatenate `s1` and `"acomb"`. Next, line 23 uses the overloaded addition assignment operator, `+=`, to concatenate `s3` and `"pet"`. Then line 29 appends the string `"comb"` to empty `string s5`. The arguments are the `std::string` to retrieve characters from (`s1`), the starting index (`4`) and the number of characters to append (`s1.size() - 4`).

8.3 Comparing **`string`**s

`std::string` provides member functions for comparing `string`s (Fig. 8.2). We call function `displayResult` (lines 7–17) throughout this example to display each comparison's result. The program declares four `string`s (lines 20–23) and outputs each (lines 25–26).

```
 1  // fig08_02.cpp
 2  // Comparing strings.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5  #include <string>
 6
 7  void displayResult(const std::string& s, int result) {
 8     if (result == 0) {
 9        std::cout << fmt::format("{} == 0\n", s);
10     }
11     else if (result > 0) {
12        std::cout << fmt::format("{} > 0\n", s);
13     }
14     else { // result < 0
```

```
15          std::cout << fmt::format("{} < 0\n", s);
16       }
17    }
18
19    int main() {
20       const std::string s1{"Testing the comparison functions."};
21       const std::string s2{"Hello"};
22       const std::string s3{"stinger"};
23       const std::string s4{s2}; // "Hello"
24
25       std::cout << fmt::format("s1: {}\ns2: {}\ns3: {}\ns4: {}",
26          s1, s2, s3, s4);
27
28       // comparing s1 and s4
29       if (s1 > s4) {
30          std::cout << "\n\ns1 > s4\n";
31       }
32
33       // comparing s1 and s2
34       displayResult("s1.compare(s2)", s1.compare(s2));
35
36       // comparing s1 (elements 2-6) and s3 (elements 0-4)
37       displayResult("s1.compare(2, 5, s3, 0, 5)",
38          s1.compare(2, 5, s3, 0, 5));
39
40       // comparing s2 and s4
41       displayResult("s4.compare(0, s2.size(), s2)",
42          s4.compare(0, s2.size(), s2));
43
44       // comparing s2 and s4
45       displayResult("s2.compare(0, 3, s4)", s2.compare(0, 3, s4));
46    }
```

```
s1: Testing the comparison functions.
s2: Hello
s3: stinger
s4: Hello

s1 > s4
s1.compare(s2) > 0
s1.compare(2, 5, s3, 0, 5) == 0
s4.compare(0, s2.size(), s2) == 0
s2.compare(0, 3, s4) < 0
```

Fig. 8.2 Comparing `strings`.

Comparing Strings with the Relational and Equality Operators

`std::string` objects may be compared to one another or to C-strings with the relational and equality operators—each returns a `bool`. Comparisons are performed **lexicographically**—that is, based on the integer values of each character. For example, `'A'` has the value 65 and `'a'` has the value 97 (see Appendix B, Character Set), so `"Apple"` would be considered less than `"apple"`, even though they are the same word. Line 29 tests whether `s1` is greater than `s4` using the overloaded > operator. In this case, `s1` starts with a capital `T`, and `s4` starts with a capital `H`. So, `s1` is greater than `s4` because `T` (84) has a higher numeric value than `H` (72).

Comparing Strings with Member Function `compare`

Line 34 compares `s1` to `s2` using `std::string` member function **`compare`**. This function returns `0` if the `string`s are equal, a positive number if `s1` is **lexicographically** greater than `s2` or a negative number if `s1` is lexicographically less than `s2`. Because a `string` starting with `'T'` is considered lexicographically greater than a string starting with `'H'`, the result is a value greater than `0`, as confirmed by the output.

The `compare` call in line 38 compares portions of `s1` and `s3` using a `compare` overload. The first two arguments (`2` and `5`) specify the starting index and length of the portion of `s1` (`"sting"`) to compare with `s3`. The third argument is the comparison `string`. The last two arguments (`0` and `5`) are the starting index and length of the portion of `s3` to compare (also `"sting"`). The two pieces being compared are identical, so `compare` returns `0` as confirmed in the output.

Line 42 uses another `compare` overload to compare `s4` and `s2`. The first two arguments are the starting index and length, and the last argument is the comparison `string`. The pieces of `s4` and `s2` being compared are identical, so `compare` returns `0`.

Line 45 compares the first `3` characters in `s2` to `s4`. Because `"Hel"` begins with the same first three letters as `"Hello"` but has fewer letters overall, `"Hel"` is considered less than `"Hello"` and `compare` returns a value less than zero.

8.4 Substrings

`std::string`'s member function **`substr`** (Fig. 8.3) returns a substring from a `string`. The result is a new `string` object with contents copied from the source `string`. Line 8 uses member function `substr` to get a substring from `s` starting at index `3` and consisting of `4` characters.

```
1   // fig08_03.cpp
2   // Demonstrating string member function substr.
3   #include <iostream>
4   #include <string>
5
6   int main() {
7      const std::string s{"airplane"};
8      std::cout << s.substr(3, 4) << '\n'; // retrieve substring "plan"
9   }
```

```
plan
```

Fig. 8.3 Demonstrating `string` member function `substr`.

8.5 Swapping `string`s

`std::string` provides member function **`swap`** for swapping (i.e., exchanging the contents of) two `string`s. Figure 8.4 calls **`swap`** (line 13) to exchange the values of `s1` and `s2`.

```
1   // fig08_04.cpp
2   // Using the swap function to swap two strings.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <string>
6
7   int main() {
```

```
 8      std::string s1{"one"};
 9      std::string s2{"two"};
10
11      std::cout << fmt::format("Before swap:\ns1: {}; s2: {}", s1, s2);
12      s1.swap(s2); // swap strings
13      std::cout << fmt::format("\n\nAfter swap:\ns1: {}; s2: {}", s1, s2);
14   }
```

```
Before swap:
s1: one; s2: two

After swap:
s1: two; s2: one
```

Fig. 8.4 Using the `swap` function to swap two `string`s.

8.6 `string` Characteristics

`std::string` provides member functions for gathering information about a `string`'s size, capacity, maximum length and other characteristics:

- A `string`'s size is the number of characters currently stored in the `string`.
- A `string`'s **capacity** is the number of characters that can be stored in the `string` before it must allocate more memory to store additional characters. A `string` performs memory allocation for you behind the scenes. The capacity of a `string` is always at least the `string`'s current size, though it can be greater. The exact capacity depends on the implementation.
- The **maximum size** is the largest possible size a `string` can have. If this value is exceeded, a `length_error` exception is thrown.[1]

---

1. In practice, the maximum size is so large in most implementations that you will not encounter an exception.

Figure 8.5 demonstrates `string` member functions for determining these characteristics. We broke the example into parts for discussion. Function `printStatistics` (lines 8–12) receives a `string` and displays its capacity (using member function **`capacity`**), maximum size (using member function **`max_size`**), size (using member function `size`) and whether the `string` is empty (using member function `empty`).

```
 1   // fig08_05.cpp
 2   // Printing string characteristics.
 3   #include <fmt/format.h>
 4   #include <iostream>
 5   #include <string>
 6
 7   // display string statistics
 8   void printStatistics(const std::string& s) {
 9      std::cout << fmt::format(
10         "capacity: {}\nmax size: {}\nsize: {}\nempty: {}",
11         s.capacity(), s.max_size(), s.size(), s.empty());
12   }
13
```

Fig. 8.5 Printing `string` characteristics.

The program declares empty `string string1` (line 15) and passes it to function `printStatistics` (line 18). This call to `printStatistics` indicates that `string1`'s initial size is `0`—it contains no characters. For Visual C++ and GNU `g++`, the maximum size (shown in the output) is 9,223,372,036,854,775,807 and for Clang in Xcode it's 18,446,744,073,709,551,599. Object `string1` is an empty `string`, so function `empty` returns `true`.

```
14   int main() {
15      std::string string1; // empty string
16
17      std::cout << "Statistics before input:\n";
18      printStatistics(string1);
19
```

```
Statistics before input:
capacity: 15
max size: 9223372036854775807
size: 0
empty: true
```

Line 21 inputs a string (we typed `tomato`). Line 24 calls `printStatistics` to output updated `string1` statistics. The size is now `6`, and `string1` is no longer empty.

```
20   std::cout << "\n\nEnter a string: ";
21   std::cin >> string1; // delimited by whitespace
22   std::cout << fmt::format("The string entered was: {}\n", string1);
23   std::cout << "Statistics after input:\n";
24   printStatistics(string1);
25
```

```
Enter a string: tomato
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 9223372036854775807
size: 6
empty: false
```

Line 27 inputs another string (we typed `soup`) and stores it in `string1`, replacing `"tomato"`. Line 30 calls `printStatistics` to output updated `string1` statistics. Note that the length is now `4`.

```
26   std::cout << "\n\nEnter a string: ";
27   std::cin >> string1; // delimited by whitespace
28   std::cout << fmt::format("The string entered was: {}\n", string1);
29   std::cout << "Statistics after input:\n";
30   printStatistics(string1);
```

```
Enter a string: soup
The string entered was: soup
Statistics after input:
capacity: 15
max size: 9223372036854775807
size: 4
empty: false
```

Line 33 uses `+=` to concatenate a 46-character string to `string1`. Line 36 calls `print-Statistics` to output updated `string1` statistics. Because `string1`'s capacity was not large enough to accommodate the new string size, the capacity was automatically increased to `63` elements, and `string1`'s size is now `50`. The manner in which the capacity grows is implementation-defined.

```
32    // append 46 characters to string1
33    string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
34    std::cout << fmt::format("\n\nstring1 is now: {}\n", string1);
35    std::cout << "Statistics after concatenation:\n";
36    printStatistics(string1);
37
```

```
string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
Statistics after concatenation:
capacity: 63
max size: 9223372036854775807
size: 50
empty: false
```

Line 38 uses member function **`resize`** to increase `string1`'s size by `10` characters. The additional elements are set to null characters. The `printStatistics` output shows that the capacity did not change, but the size is now `60`.

```
38       string1.resize(string1.size() + 10); // add 10 elements to string1
39       std::cout << "\n\nStatistics after resizing to add 10 characters:\n";
40       printStatistics(string1);
41       std::cout << '\n';
42    }
```

```
Statistics after resizing to add 10 characters:
capacity: 63
max size: 9223372036854775807
size: 60
empty: false
```

C++20 Update to `string` Member-Function `reserve`

20 You can change the capacity of a `string` without changing its size by calling `string` member function **`reserve`**. If its integer argument is greater than the current capacity, the capacity is increased to greater than or equal to the argument value. As of C++20, if `reserve`'s argument is less than the current capacity, the capacity does not change. Before C++20, `reserve` could reduce the capacity and, if the argument were smaller than the `string`'s size, could reduce the capacity to match the size.

8.7 Finding Substrings and Characters in a `string`

`std::string` provides member functions for finding substrings and characters in a `string`. Figure 8.6 demonstrates the `find` functions. We broke this example into parts for discussion. String `s` is declared and initialized in line 8.

```
1   // fig08_06.cpp
```

```
2   // Demonstrating the string find member functions.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <string>
6
7   int main() {
8      const std::string s{"noon is 12pm; midnight is not"};
9      std::cout << "Original string: " << s;
10
```

```
Original string: noon is 12pm; midnight is not
```

Fig. 8.6 Demonstrating the string find member functions.

Member Functions `find` and `rfind`

Lines 12–13 attempt to find `"is"` in `s` using member functions **`find`** and **`rfind`**, which search from the beginning and end of `s`, respectively. If `"is"` is found, the index of the starting location of that string is returned. If the `string` is not found, the `string find`-related functions return the constant **`string::npos`** to indicate that a substring or character was not found in the `string`. The rest of the `find` functions presented in this section return the same type unless otherwise noted.

```
11   // find "is" from the beginning and end of s
12   std::cout << fmt::format("\ns.find(\"is\"): {}\ns.rfind(\"is\"): {}",
13      s.find("is"),s.rfind("is"));
14
```

```
s.find("is"): 5
s.rfind("is"): 23
```

Member Function `find_first_of`

Line 16 uses member function **`find_first_of`** to locate the first occurrence in `s` of any character in `"misop"`. The searching is done from the beginning of `s`. The character `'o'` is found at index 1.

```
15   // find 'o' from beginning
16   int location{s.find_first_of("misop")};
17   std::cout << fmt::format("\ns.find_first_of(\"misop\") found {} at {}",
18      s.at(location), location);
```

```
s.find_first_of("misop") found o at 1
```

Member Function `find_last_of`

Line 21 uses member function **`find_last_of`** to find the last occurrence in `s` of any character in `"misop"`. The searching is done from the end of `s`. The character `'o'` is found at index 27 of the string.

```
20   // find 'o' from end
21   location = s.find_last_of("misop");
22   std::cout << fmt::format("\ns.find_last_of(\"misop\") found {} at {}",
23      s.at(location), location);
```

```
s.find_last_of("misop") found o at 27
```

Member Function `find_first_not_of`

Line 26 uses member function **`find_first_not_of`** to find the first character from the beginning of `s` that is not contained in `"noi spm"`, finding `'1'` at index 8. Line 32 uses member function `find_first_not_of` to find the first character not contained in `"12noi spm"`. It searches from the beginning of `s` and finds `';'` at index 12. Line 38 uses member function `find_first_not_of` to find the first character not contained in `"noon is 12pm; midnight is not"`. In this case, the `string` being searched contains every character specified in the string argument. Because a character was not found, `string::npos` (which has the value `-1` in this case) is returned.

```
25      // find '1' from beginning
26      location = s.find_first_not_of("noi spm");
27      std::cout << fmt::format(
28         "\ns.find_first_not_of(\"noi spm\") found {} at {}",
29         s.at(location), location);
30
31      // find ';' at location 12
32      location = s.find_first_not_of("12noi spm");
33      std::cout << fmt::format(
34         "\ns.find_first_not_of(\"12noi spm\") found {} at {}",
35         s.at(location), location);
36
37      // search for characters not in "noon is 12pm; midnight is not"
38      location = s.find_first_not_of("noon is 12pm; midnight is not");
39      std::cout << fmt::format("\n{}: {}\n",
40         "s.find_first_not_of(\"noon is 12pm; midnight is not\")",
41         location);
42   }
```

```
s.find_first_not_of("noi spm") found 1 at 8
s.find_first_not_of("12noi spm") found ; at 12
s.find_first_not_of("noon is 12pm; midnight is not"): -1
```

8.8 Replacing and Erasing Characters in a `string`

Figure 8.7 demonstrates `string` member functions for replacing and erasing characters. We broke this example into parts for discussion. Lines 9–13 declare and initialize `string1`. When string literals are separated only by whitespace, the compiler concatenates them into a single string literal. Breaking apart lengthy strings in this manner can make your code more readable.

```
1   // fig08_07.cpp
2   // Demonstrating string member functions erase and replace.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <string>
6
7   int main() {
8   // compiler concatenates all parts into one string
9   std::string string1{"The values in any left subtree"
10     "\nare less than the value in the"
11     "\nparent node and the values in"
12     "\nany right subtree are greater"
```

```
      "\nthan the value in the parent node"};

   std::cout << fmt::format("Original string:\n{}\n\n", string1);

Original string:
The values in any left subtree
are less than the value in the
parent node and the values in
any right subtree are greater
than the value in the parent node
```

Fig. 8.7 Demonstrating `string` member functions `erase` and `replace`.

Line 17 uses `string` member function **`erase`** to erase everything from (and including) the character in position 62 to the end of `string1`. Each newline character occupies one character in the `string`.

```
   string1.erase(62); // remove from index 62 through end of string1
   std::cout << fmt::format("string1 after erase:\n{}\n\n", string1);

string1 after erase:
The values in any left subtree
are less than the value in the
```

Lines 20–26 use `find` to locate each occurrence of the space character. Each space is then replaced with a period by a call to `string` member function **`replace`**, which takes three arguments:

- the index of the character in the `string` at which replacement should begin,
- the number of characters to replace and
- the replacement string.

Member function `find` returns `string::npos` when the search character is not found. In line 25, we add `1` to `position` to continue searching from the next character's location.

```
   size_t position{string1.find(" ")}; // find first space

   // replace all spaces with period
   while (position != std::string::npos) {
      string1.replace(position, 1, ".");
      position = string1.find(" ", position + 1);
   }

   std::cout << fmt::format("After first replacement:\n{}\n\n", string1);

After first replacement:
The.values.in.any.left.subtree
are.less.than.the.value.in.the
```

Lines 30–37 use functions `find` and `replace` to find every period and replace every period and its following character with two semicolons. The arguments passed to this version of `replace` are

- the index of the element where the replace operation begins,
- the number of characters to replace,
- a replacement character string from which a substring is selected to use as replacement characters,
- the element in the character string where the replacement substring begins and
- the number of characters in the replacement character string to use.

```
30       position = string1.find("."); // find first period
31
32       // replace all periods with two semicolons
33       // NOTE: this will overwrite characters
34       while (position != std::string::npos) {
35          string1.replace(position, 2, "xxxxx;;yyy", 5, 2);
36          position = string1.find(".", position + 2);
37       }
38
39       std::cout << fmt::format("After second replacement:\n{}\n", string1);
40    }
```

```
After second replacement:
The;;alues;;n;;ny;;eft;;ubtree
are;;ess;;han;;he;;alue;;n;;he
```

8.9 Inserting Characters into a `string`

`std::string` provides overloaded member functions for inserting characters into a `string` (Fig. 8.8). Line 14 uses `string` member function **`insert`** to insert `"middle "` before index 10 of `s1`. Line 15 uses `insert` to insert `"xx"` before `s2`'s index 3. The last two arguments specify the starting and last element of `"xx"` to insert. Using `string::npos` causes the entire `string` to be inserted.

```
1   // fig08_08.cpp
2   // Demonstrating std::string insert member functions.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <string>
6
7   int main() {
8      std::string s1{"beginning end"};
9      std::string s2{"12345678"};
10
11     std::cout << fmt::format("Initial strings:\ns1: {}\ns2: {}\n\n",
12        s1, s2);
13
14     s1.insert(10, "middle "); // insert "middle " at location 10
15     s2.insert(3, "xx", 0, std::string::npos); // insert "xx" at location 3
16
17     std::cout << fmt::format("Strings after insert:\ns1: {}\ns2: {}\n",
18        s1, s2);
19  }
```

```
Initial strings:
s1: beginning end
s2: 12345678

Strings after insert:
s1: beginning middle end
s2: 123xx45678
```

Fig. 8.8 Demonstrating `std::string insert` member functions.

8.10 C++11 Numeric Conversions

C++11 added functions for converting from numeric values to `string`s and from `string`s to numeric values.

Converting Numeric Values to `string` Objects

C++11's `to_string` function (from header `<string>`) returns the `string` representation of its numeric argument. The function is overloaded for the fundamental numeric types `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float`, `double` and `long double`.

Converting `string` Objects to Numeric Values

C++11 provides eight functions (from the `<string>` header) for converting `string` objects to numeric values:

Function	Return type
Functions that convert to integral types	
`stoi`	`int`
`stol`	`long`
`stoul`	`unsigned long`
`stoll`	`long long`
`stoull`	`unsigned long long`
`stof`	`float`
`stod`	`double`
`stold`	`long double`

Each function attempts to convert the beginning of its `string` argument to a numeric value. If no conversion can be performed, each function throws an `invalid_argument` exception. If the conversion result is out of range for the function's return type, each function throws an `out_of_range` exception.

Functions That Convert `string`s to Integral Types

Consider an example of converting a `string` to an integral value. Assuming the `string`

```
string s{"100hello"};
```

the following statement converts the beginning of the string to the `int` value `100` and stores that value in `convertedInt`:

```
int convertedInt{stoi(s)};
```

Each function that converts a `string` to an integral type receives three parameters—the last two have default arguments. The parameters are

- A `string` containing the characters to convert.
- A pointer to a `size_t` variable. The function uses this pointer to store the index of the first character that was not converted. The default argument is `nullptr`, in which case the function does not store the index.
- An `int` that's either 0 or a value in the range 2–36 representing the number's base—the default is base 10. If the base is 0, the function auto-detects the base.

So, the preceding statement is equivalent to

```
int convertedInt{stoi(s, nullptr, 10)};
```

Given a `size_t` variable named `index`, the statement

```
int convertedInt{stoi(s, &index, 2)};
```

converts the binary number `"100"` (base 2) to an `int` (100 in binary is the `int` value 4) and stores in `index` the location of the letter `"h"` (the first character that was not converted).

Functions That Convert `string`s to Floating-Point Types

The functions that convert `string`s to floating-point types each receive two parameters:

- A `string` containing the characters to convert.
- A pointer to a `size_t` variable where the function stores the index of the first character that was not converted. The default argument is `nullptr`, in which case the function does not store the index.

Consider an example of converting the following `string` to a floating-point value:

```
string s{"123.45hello"};
```

The statement

```
double convertedDouble{stod(s)};
```

converts the beginning of `s` to the `double` value `123.45` and stores that value in the variablen `convertedDouble`. Again, the second argument is `nullptr` by default.

17 8.11 C++17 `string_view`

C++17 introduced **`string_view`s** (header **`<string_view>`**), which are read-only views of the contents of C-strings or `string` objects. They also can view a range of characters in a container, such as an `array` or `vector` of `char`s. Like `std::span`, a `string_view` does not own the data it views. It contains

- a pointer to the first character in a contiguous sequence of characters and
- a count of the number of characters.

A `string_view` is not automatically updated if the contents it views change in size after the `string_view` is initialized.

Perf CG CG `string_view`s enable many `string`-style operations on C-strings without the overhead of creating and initializing `string` objects, which copies the C-string contents. The C++ Core Guidelines state that you should prefer `string` objects if you need to "own character sequences"—for example, so you can modify a `string`'s contents.[2] If you simply need a read-only view of a contiguous sequence of characters, the C++ Core Guidelines recommend using a `string_view`.[3]

2. C++ Core Guidelines, "SL.str.1: Use `std::string` to Own Character Sequences." Accessed January 3, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-string.
3. C++ Core Guidelines, "SL.str.2: Use `std::string_view or gsl::span<char>` to Refer to Character Sequences." Accessed January 3, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rstr-view.

Creating a `string_view`

Figure 8.9 demonstrates several `string_view` features. We broke this example into parts for discussion. Line 6 includes the header `<string_view>`. Line 9 creates the `string s1`, and line 10 copies `s1` into the `string s2`. Line 11 uses `s1` to initialize a `string_view`.

```
1   // fig08_09.cpp
2   // C++17 string_view.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <string>
6   #include <string_view>
7
8   int main() {
9      std::string s1{"red"};
10     std::string s2{s1};
11     std::string_view v1{s1}; // v1 "sees" the contents of s1
12     std::cout << fmt::format("s1: {}\ns2: {}\nv1: {}\n\n", s1, s2, v1);
13
```

```
s1: red
s2: red
v1: red
```

Fig. 8.9 C++17 `string_view`.

`string_views` "See" Changes to the Characters They View

Because a `string_view` does not own the sequence of characters it views, it "sees" any changes to the original characters. Line 15 modifies the `std::string s1`. Then line 16 shows `s1`'s, `s2`'s and the `string_view v1`'s contents. Note that `s2` was not modified because it owns a copy of `s1`'s contents.

```
14  // string_views see changes to the characters they view
15  s1.at(0) = 'R'; // capitalize s1
16  std::cout << fmt::format("s1: {}\ns2: {}\nv1: {}\n\n", s1, s2, v1);
17
```

```
s1: Red
s2: red
```

```
v1: Red
```

`string_view`s Are Comparable to `std::string`s or `string_view`s

Like `string`s, `string_view`s support the relational and equality operators. You also can intermix `std::string`s and `string_view`s as in line 20's `==` comparisons.

```
18    // string_views are comparable with strings or string_views
19    std::cout << fmt::format("s1 == v1: {}\ns2 == v1: {}\n\n",
20       s1 == v1, s2 == v1);
```

```
s1 == v1: true
s2 == v1: false
```

`string_view`s Can Remove a Prefix or Suffix

Perf You can easily remove a specified number of characters from the beginning or end of a `string_view`. These are fast operations for a `string_view`—they simply adjust the character count and, in the case of removing from the beginning, move the pointer to the first character in the `string_view`. Lines 23–24 call `string_view` member functions **`remove_prefix`** and **`remove_suffix`** to remove one character from the beginning and one from the end of `v1`, respectively. Note that `s1` remains unmodified.

```
22    // string_view can remove a prefix or suffix
23    v1.remove_prefix(1); // remove one character from the front
24    v1.remove_suffix(1); // remove one character from the back
25    std::cout << fmt::format("s1: {}\nv1: {}\n\n", s1, v1);
```

```
s1: Red
v1: e
```

`string_view`s Are Iterable

Line 28 initializes a `string_view` from a C-string. Like `string`s, `string_view`s are iterable, so you can use them with the range-based `for` statement, as in lines 30–32.

```
27    // string_views are iterable
28    std::string_view v2{"C-string"};
29    std::cout << "The characters in v2 are: ";
30    for (char c : v2) {
31       std::cout << c << " ";
32    }
```

```
The characters in v2 are: C - s t r i n g
```

`string_view`s Enable Various String Operations on C-Strings

Many `string` member functions that do not modify a `string`'s contents also are defined for `string_view`s. For example,

- line 35 calls **`size`** to determine the number of characters the `string_view` can "see,"
- line 36 calls **`find`** to get the index of `'-'` in the `string_view` and

- lines 37–38 use the new C++20 `starts_with` function to determine whether the `string_view` starts with `'C'`.

For a complete list of `string_view` member functions, see

```
https://en.cppreference.com/w/cpp/string/basic_string_view
```

```
34      // string_views enable various string operations on C-Strings
35      std::cout << fmt::format("\n\nv2.size(): {}\n", v2.size());
36      std::cout << fmt::format("v2.find('-'): {}\n", v2.find('-'));
37      std::cout << fmt::format("v2.starts_with('C'): {}\n",
38         v2.starts_with('C'));
39   }
v2.size(): 8
v2.find('-'): 1
v2.starts_with('C'): true
```

8.12 Files and Streams

C++ views each file simply as a sequence of bytes:

Each file ends either with an **end-of-file marker** or at a specific byte number recorded in an operating-system-maintained administrative data structure. When a file is opened, an object is created, and a stream is associated with the object. The objects `cin`, `cout`, `cerr` and `clog` are created for you in the header `<iostream>`. The streams associated with these objects provide communication channels between a program and a particular file or device. The `cin` object (standard input stream object) enables a program to input data from the keyboard or other devices. The `cout` object (standard output stream object) enables a program to output data to the screen or other devices. The objects `cerr` and `clog` (standard error stream objects) enable a program to output error messages to the screen or other devices. Messages written to `cerr` are output immediately. In contrast, messages written to `clog` are stored in a memory object called a buffer. When the buffer is full (or flushed; see online Chapter 19), its contents are written to the standard error stream.

File-Processing Streams

File processing requires header `<fstream>`, which includes the following definitions:

- `ifstream` is for file input.
- `ofstream` is for file output.
- `fstream` combines the capabilities of `ifstream` and `ofstream`.

The `cout` and `cin` capabilities we've discussed so far and the additional I/O features we describe in online Chapter 19 also can be applied to file streams.

8.13 Creating a Sequential File

C++ imposes no structure on files. Thus, a concept like that of a record containing related data items does not exist. You structure files to meet your application's requirements. The following example shows how to impose a simple record structure on a file.

Figure 8.10 creates a sequential file that might be used in an accounts-receivable system to help keep track of the money owed to a company by its credit clients. For each client, the program obtains the client's account number, name and balance (i.e., the amount the client owes the company for goods and services received in the past). The data obtained for each client constitutes a record for that client.

```
1   // fig08_10.cpp
2   // Creating a sequential file.
3   #include <cstdlib> // exit function prototype
4   #include <fmt/format.h>
5   #include <fstream> // contains file stream processing types
6   #include <iostream>
7   #include <string>
8
9   int main() {
10     // ofstream opens the file
11     if (std::ofstream output{"clients.txt", std::ios::out}) {
12        std::cout << "Enter the account, name, and balance.\n"
13           << "Enter end-of-file to end input.\n? ";
14
15        int account;
16        std::string name;
17        double balance;
18
19        // read account, name and balance from cin, then place in file
20        while (std::cin >> account >> name >> balance) {
21           output << fmt::format("{} {} {}\n", account, name, balance);
22           std::cout << "? ";
23        }
24     }
25     else {
26        std::cerr << "File could not be opened\n";
27        std::exit(EXIT_FAILURE);
28     }
29  }
```

```
Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Fig. 8.10 Creating a sequential file.

Opening a File

Err Figure 8.10 writes data to a file, so we open the file for output by creating an `ofstream` object (line 11) and initializing it with the **filename** and the **file-open mode**. The file-open mode **`ios::out`** is the default for an `ofstream` object, so the second argument in line 11 is not required. Use caution when opening an existing file for output (`ios::out`). Existing files opened with mode `ios::out` are **truncated**—all data in the file is discarded without warning. If the file does not exist, the `ofstream` object creates the file.

11 17 Line 11 creates the `ofstream` object `output` associated with the file `clients.txt` and opens it for output. We did not specify a path to the file on disk, so it's placed in the same folder as the program's executable file. Before C++11, the filename was specified as a pointer-based string. C++11 added specifying the filename as a `string` object. C++17 introduced the **`<filesystem>` header** with features for manipulating files and folders. So, you also may specify the file to open as a **`filesystem::path`** object.

The following table lists the file-open modes. These modes can be combined by separating them with the `|` operator:

Mode	Description
`ios::app`	Append all output to the end of the file without modifying any data already in the file.
`ios::ate`	Open a file for output and move to the end of the file. Used to append data to a file. Data can be written anywhere in the file.
`ios::in`	Open a file for input.
`ios::out`	Open a file for output.
`ios::trunc`	Discard the file's contents. This is the default action for ios::out.
`ios::binary`	Open a file for binary (i.e., non-text) input or output.

Opening a File via the `open` Member Function

You can create an `ofstream` object without opening a specific file. In this case, a file can be attached to the object later. For example, the statement

```
ofstream output;
```

creates an `ofstream` object that's not yet associated with a file. The `ofstream` member function **`open`** opens a file and attaches it to an existing `ofstream` object as follows:

```
output.open("clients.txt", ios::out);
```

Again, `ios::out` is the default value for the second argument.

Testing Whether a File Was Opened Successfully

11 After creating an `ofstream` object and attempting to open the file, the `if` statement uses the file object `output` as a condition (line 11) to determine whether the `open` operation succeeded. For a file object, there is an overloaded `operator bool` (added in C++11) that implicitly evaluates the file object to `true` if the file opened successfully or `false` otherwise. Some possible reasons opening a file might fail are

- attempting to open a nonexistent file for reading,
- Sec attempting to open a file for reading or writing in a directory that you don't have permission to access and
- opening a file for writing when no secondary storage space is available.

If the condition indicates an unsuccessful attempt to open the file, line 26 outputs an error message, and line 27 invokes function `exit` to terminate the program. The argument to `exit` is returned to the environment from which the program was invoked. Passing `EXIT_SUCCESS` (defined in `<cstdlib>`) to `exit` indicates that the program terminated normally; passing any other value (in this case, `EXIT_FAILURE`) indicates that the program terminated due to an error.[4]

4. In `main`, you can simply return `EXIT_SUCCESS` or `EXIT_FAILURE`, rather than calling `std::exit`. When `main` terminates, its local variables are destroyed, then `main` calls `std::exit`, passing the value in `main's return` statement (https://timsong-cpp.github.io/cppwp/n4861/basic.start.main#5). Recall that `main` implicitly returns `0` to indicate successful execution if you do not specify a `return` statement.

Processing Data

If line 11 opens the file successfully, the program begins processing data. Lines 12–13 prompt the user to enter the various fields for each record or the end-of-file indicator if data entry is complete. To enter the end-of-file indicator on Linux or macOS, type *<Ctrl-d>* on a line by itself. On Microsoft Windows, type *<Ctrl-z>* then press *Enter*

The `while` statement's condition (line 20) implicitly invokes the `operator bool` function on `cin`. The condition remains `true` as long as each input operation with `cin` is successful. Entering the end-of-file indicator causes the `operator bool` member function to return `false`. You also can call member function `eof` on the `input` object to determine whether the end-of-file indicator has been entered.

Line 20 extracts each set of data into the variables `account`, `name` and `balance`, and determines whether the end-of-file indicator has been entered. When end-of-file is encountered (that is, when the user enters the end-of-file key combination) or an input operation fails, the `operator bool` returns `false`, and the `while` statement terminates.

Line 21 writes a set of data to the file `clients.txt`, using the stream insertion operator `<<` and the `output` object we associated with the file at the beginning of the program. The data may be retrieved by a program designed to read the file (see Section 8.14). The file created in Fig. 8.10 is simply a text file that can be viewed by any text editor.

Closing a File

Once the user enters the end-of-file indicator, the `while` loop terminates. At this point, we reach the `if` statement's closing brace, so the `output` object goes out of scope, which automatically closes the file. You should always close a file as soon as it's no longer needed in a program. You also can close a file object explicitly, using member function **`close`**, as in

```
output.close();
```

Sample Execution

In the sample execution of Fig. 8.10, we entered information for five accounts, then signaled that data entry was complete by entering the end-of-file indicator (`^Z` is displayed for Microsoft Windows). This dialog window does not show how the data records appear in the file. The next section shows how to create a program that reads this file and prints its contents.

8.14 Reading Data from a Sequential File

Figure 8.10 created a sequential file. Figure 8.11 reads data sequentially from the file `clients.txt` and displays the records. Creating an `ifstream` object opens a file for input. An `ifstream` is initialized with a filename and file-open mode. Line 11 creates an `ifstream` object called `input` that opens the `clients.txt` file for reading.[5] If a file's contents should not be modified, use `ios::in` to open it only for input to prevent unintentional modification of the file's contents.

5. The file `clients.txt` must be in the same folder as this program's executable.

```
1   // fig08_11.cpp
2   // Reading and printing a sequential file.
3   #include <cstdlib>
4   #include <fmt/format.h>
5   #include <fstream> // file stream
6   #include <iostream>
7   #include <string>
8
9   int main() {
10     // ifstream opens the file
11     if (std::ifstream input{"clients.txt", std::ios::in}) {
12        std::cout << fmt::format("{:<10}{:<13}{:>7}\n",
13           "Account", "Name", "Balance");
14
15        int account;
16        std::string name;
17        double balance;

18
19        // display each record in file
20        while (input >> account >> name >> balance) {
21           std::cout << fmt::format("{:<10}{:<13}{:>7.2f}\n",
22              account, name, balance);
23        }
24     }
25     else {
26        std::cerr << "File could not be opened\n";
27        std::exit(EXIT_FAILURE);
28     }
29  }
```

```
Account   Name        Balance
100       Jones         24.98
200       Doe          345.67
300       White          0.00
400       Stone        -42.16
500       Rich         224.62
```

Fig. 8.11 Reading and printing a sequential file.

Opening a File for Input

Objects of class `ifstream` are opened for input by default, so you can omit `ios::in` when you create the `ifstream`, as in

```
std::ifstream input{"clients.txt"};
```

An `ifstream` object can be created without opening a file—you can attach one to it later. Before attempting to retrieve data from the file, line 11 uses the `input` object as a condition to determine whether the file was opened successfully. The overloaded operator `bool` returns a `true` if the file was opened; otherwise, it returns `false`.

Reading from the File

Line 20 reads a set of data (i.e., a record) from the file. After line 20 executes the first time, `account` has the value `100`, `name` has the value `"Jones"` and `balance` has the value `24.98`. Each time line 20 executes, it reads another record into the variables `account`, `name` and `balance`. Lines 21–22 display each record. When the end of the file is reached, the implicit call to `operator bool` in the `while` condition returns `false`, the `ifstream` object goes out of scope at line 24 (which automatically closes the file) and the program terminates.

File-Position Pointers

Programs often read sequentially from the beginning of a file and read all the data consecutively until the desired data is found. It might be necessary to process the file sequentially several times (from the beginning) during the execution of a program. `istream` and `ostream` provide member functions **`seekg`** ("seek get") and **`seekp`** ("seek put") to reposition the **file-position pointer**. This represents the byte number of the next byte in the file to be read or written. Each `istream` object has a **get pointer**, which indicates the byte number in the file from which the next input is to occur. Each `ostream` object has a **put pointer**, which indicates the byte number in the file at which the next output should be placed. The statement

```
input.seekg(0);
```

repositions the *get* file-position pointer to the beginning of the file (location `0`) attached to `input`. The argument to `seekg` is an integer. If the end-of-file indicator has been set, you'd also need to execute

```
input.clear();
```

to re-enable reading from the stream.

An optional second argument indicates the **seek direction**:

- `ios::beg` (the default) for positioning relative to the beginning of a stream,
- `ios::cur` for positioning relative to the current position in a stream or
- `ios::end` for positioning backward relative to the end of a stream.

When seeking from the beginning of a file, the file-position pointer is an integer value that specifies a location as a number of bytes from the file's starting location. This is also referred to as the **offset** from the beginning of the file. Some examples of moving the *get* file-position pointer are

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg(n);

// position n bytes forward from the current position in fileObject
fileObject.seekg(n, ios::cur);

// position n bytes back from end of fileObject
fileObject.seekg(n, ios::end);

// position at end of fileObject
fileObject.seekg(0, ios::end);
```

The same operations can be performed using `ostream` member function `seekp`. Member functions **`tellg`** and **`tellp`** return the current locations of the *get* and *put* pointers, respectively. The following statement assigns the *get* file-position pointer value to variable `location` of type `std::istream::pos_type`:

```
auto location{fileObject.tellg()};
```

14 8.15 C++14 Reading and Writing Quoted Text

Many text files contain quoted text, such as `"C++20 for Programmers"`. For example, in files representing HTML5 webpages, attribute values are enclosed in quotes. If you're building a web browser to display the contents of such a webpage, you must be able to read those quoted strings and remove the quotes.

Suppose you need to read from a text file, as you did in Fig. 8.11, but with each account's data formatted as follows:

```
100 "Janie Jones" 24.98
```

Recall that the stream extraction operator `>>` treats whitespace as a delimiter. So, if we read the preceding data using the expression in line 20 of Fig. 8.11:

```
input >> account >> name >> balance
```

the first stream extraction reads `100` into the `int` variable `account`, and the second reads only `"Janie` into the `string` variable `name`. The opening double quote would be part of the `string` in `name`. The third stream extraction fails while attempting to read a value for the `double` variable `balance` because the next token (i.e., piece of data) in the input stream—`Jones"`—is not a `double`.

Reading Quoted Text

14 C++14 added the stream manipulator **`quoted`** (header `<iomanip>`) for reading quoted text from a stream. It includes any whitespace characters in the quoted text and discards the double-quote delimiters. For example, assuming the data

```
100 "Janie Jones" 24.98
```

the expression

```
input >> account >> std::quoted(name) >> balance
```

reads `100` into `account`, reads `Janie Jones` as one `string` into `name` and reads `24.98` into `balance`. If the quoted data contains `\"` or `\\` escape sequences, each is read and stored in the `string` as `"` or `\`, respectively.

Writing Quoted Text

Similarly, you can write quoted text to a stream. For example, if `name` contains `Janie Jones`, the statement

```
outputStream << std::quoted(name);
```

writes to the `outputStream`

```
"Janie Jones"
```

If the string contains a `"` or `\`, it will be displayed as `\"` or `\\`.

8.16 Updating Sequential Files

Data that is formatted and written to a sequential file, as shown in Section 8.13, cannot be modified in place without the risk of destroying other data in the file. For example, if the name "`White`" needs to be changed to "`Worthington`," the old name cannot be overwritten without corrupting the file. The record for `White` was written to the file as

```
300 White 0.00
```

If this record were rewritten beginning at the same location in the file using the longer name, the record would be

```
300 Worthington 0.00
```

The new record contains six more characters than the original. Any characters after "`h`" in "`Worthington`" would overwrite `0.00` and the beginning of the next sequential record in the file. The problem with the formatted input/output model using the stream insertion operator `<<` and the stream extraction operator `>>` is that fields—and hence records—can vary in size. For example, values 7, 14, –117, 2074, and 27383 are all `int`s, which store the same number of "raw data" bytes internally. However, these integers become different-sized fields, depending on their actual values, when output as formatted text (character sequences). Therefore, the formatted input/output model usually is not used to update records in place.

Such updating can be done with sequential files, but awkwardly. For example, to make the preceding name change in a sequential file, we could:

- copy the records before `300 White 0.00` to a new file,
- write the updated record to the new file, then
- write the records after `300 White 0.00` to the new file.

Then we could delete the old file and rename the new one. This requires processing every record in the file to update one record. If many records are being updated in one pass of the file, though, this technique can be acceptable.

8.17 String Stream Processing

In addition to standard stream I/O and file stream I/O, C++ includes capabilities for inputting from and outputting to `string`s in memory. These capabilities often are referred to as **in-memory I/O** or **string stream processing**. You can read from a `string` with **`istringstream`** and write to a `string` with **`ostringstream`**, both from the header **`<sstream>`**.

Class templates `istringstream` and `ostringstream` provide the same functionality as classes `istream` and `ostream` plus other member functions specific to in-memory formatting. An `ostringstream` object uses a `string` object to store the output data. Its **`str`** member function returns a copy of that `string`.

One application of string stream processing is data validation. A program can read an entire line at a time from the input stream into a `string`. Next, a validation routine can scrutinize the `string`'s contents and correct (or repair) the data, if necessary. Then the program can input from the `string`, knowing that the input data is in the proper format.

11 To assist with data validation, C++11 added powerful pattern-matching regular-expression capabilities. For instance, in a program requiring a U.S. format telephone number (e.g., `(800) 555-1212`), you can use a regular expression to confirm that a string matches that format. Many websites provide regular expressions for validating e-mail addresses, URLs, phone numbers, addresses and other popular kinds of data. We introduce regular expressions and provide several examples in Section 8.20.

Demonstrating `ostringstream`

Figure 8.12 creates an `ostringstream` object, then uses the stream insertion operator to output a series of `string`s and numerical values to the object.

```
1  // fig08_12.cpp
2  // Using an ostringstream object.
3  #include <iostream>
4  #include <sstream> // header for string stream processing
5  #include <string>
6
7  int main() {
8     std::ostringstream output; // create ostringstream object
9
10    const std::string string1{"Output of several data types "};
11    const std::string string2{"to an ostringstream object:"};
12    const std::string string3{"\ndouble: "};
13    const std::string string4{"\n int: "};
14
15    constexpr double d{123.4567};
```

```
16      constexpr int i{22};
17
18      // output strings, double and int to ostringstream
19      output << string1 << string2 << string3 << d << string4 << i;
20
21      // call str to obtain string contents of the ostringstream
22      std::cout << "output contains:\n" << output.str();
23
24      // add additional characters and call str to output string
25      output << "\nmore characters added";
26      std::cout << "\n\noutput now contains:\n" << output.str() << '\n';
27   }
```

```
output contains:
Output of several data types to an ostringstream object:
double: 123.457
   int: 22

output now contains:
Output of several data types to an ostringstream object:
double: 123.457
   int: 22
more characters added
```

Fig. 8.12 Using an `ostringstream` object.

Line 19 outputs `string string1`, `string string2`, `string string3`, `double d`, `string string4` and `int i`—all to `output` in memory. Line 22 displays `output.str()`, which returns the `string` created by `output` in line 19. Line 25 appends more data to the `string` in memory by simply issuing another stream insertion operation to `output`, then line 26 displays the updated contents.

Demonstrating `istringstream`

An `istringstream` object inputs data from a `string` in memory. Data is stored in an `istringstream` object as characters. Input from the `istringstream` object works identically to input from any file. The end of the `string` is interpreted by the `istringstream` object as end-of-file.

Figure 8.13 demonstrates input from an `istringstream` object. Line 9 creates `string inputString`, which consists of characters representing two strings (`"Amanda"` and `"test"`), an `int` (`123`), a `double` (`4.7`) and a `char` (`'A'`). Line 10 creates the `istringstream` object `input` and initializes it to read from `inputString`. The data items in `input-String` are read into variables `s1`, `s2`, `i`, `d` and `c` in line 17, then displayed in lines 19–20. Next, we attempt to read from `input` again in line 23, but the operation fails because there is no more data in `inputString` to read. So the `input` object evaluates to `false`, and the `else` part of the `if` ... `else` statement executes.

```
1   // fig08_13.cpp
2   // Demonstrating input from an istringstream object.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <sstream>
6   #include <string>
7
8   int main() {
9      const std::string inputString{"Amanda test 123 4.7 A"};
10     std::istringstream input{inputString};
11     std::string s1;
12     std::string s2;
```

```
      int i;
      double d;
      char c;

      input >> s1 >> s2 >> i >> d >> c;

      std::cout << "Items extracted from the istringstream object:\n"
         << fmt::format("{}\n{}\n{}\n{}\n{}\n", s1, s2, i, d, c);

      // attempt to read from empty stream
      if (long value; input >> value) {
         std::cout << fmt::format("\nlong value is: {}\n", value);
      }
      else {
         std::cout << fmt::format("\ninput is empty\n");
      }
   }
```

```
Items extracted from the istringstream object:
Amanda
test
123
4.7
A

input is empty
```

Fig. 8.13 Demonstrating input from an `istringstream` object.

8.18 Raw String Literals

Recall that backslash characters in strings introduce escape sequences—like `\n` for newline and `\t` for tab. If you wish to include a backslash in a string, you must use two backslash characters `\\`, making some strings difficult to read. For example, Microsoft Windows uses backslashes to separate folder names when specifying a file's location. To represent a file's location on Windows, you might write

```
std::string windowsPath{"C:\\MyFolder\\MySubFolder\\MyFile.txt"};
```

11 For such cases, **raw string literals** (introduced in C++11), which have the format

```
R"(rawCharacters)"
```

are more convenient. The parentheses are required around the *rawCharacters* that compose the raw string literal. The compiler automatically inserts backslashes as necessary in a raw string literal to properly escape special characters like double quotes (`"`), backslashes (`\`), etc. Using a raw string literal, we can write the preceding `string` as:

```
std::string windowsPath{R"(C:\MyFolder\MySubFolder\MyFile.txt)"};
```

Raw strings can make your code more readable, particularly when using the regular expressions that we discuss in Section 8.20. Regular expressions often contain many backslash characters.

A raw string literal can include optional delimiters up to 16 characters long before the left parenthesis, `(`, and after the right parenthesis, `)`, as in

```
R"MYDELIMITER(J.*\d[0-35-9]-\d\d-\d\d)MYDELIMITER"
```

The optional delimiters must be identical if provided. The optional delimiters are required if the raw string literal might contain one or more right parentheses. Otherwise, the first right parenthesis encountered would be treated as the end of the raw string literal.

Raw string literals may be used in any context that requires a string literal. They may also include line breaks, in which case the compiler inserts newline characters. For example, the raw string literal

```
R"(multiple lines
of text)"
```

is treated as the string literal

```
"multiple lines\nof text"
```

Caution: Any indentation in the second and subsequent lines of the raw string literal is included in the raw string literal.

8.19 Objects-Natural Case Study: Reading and Analyzing a CSV File Containing *Titanic* Disaster Data

The **CSV (comma-separated values)** file format, which uses the **`.csv` file extension**, is particularly popular, especially for datasets used in big data, data analytics and data science, and in artificial intelligence applications like machine learning and deep learning. Here, we'll demonstrate reading from a CSV file.

Datasets

There's an enormous variety of free datasets available online. The OpenML machine learning resource site

```
https://openml.org
```

contains over 21,000 free datasets in CSV format. Another great source of datasets is:

```
https://github.com/awesomedata/awesome-public-datasets
```

`account.csv`

For our first example, we've included a simple dataset in `accounts.csv` in the `ch08` folder. This file contains the account information shown in Fig. 8.11's output, but in the following format:

```
account,name,balance
100,Jones,24.98
200,Doe,345.67
300,White,0.0
```

```
400,Stone,-42.16
500,Rich,224.62
```

The first row of a CSV file typically contains column names. Each subsequent row contains one data record representing the values for those columns. In this dataset, we have three columns representing an `account`, `name` and `balance`.

8.19.1 Using `rapidcsv` to Read the Contents of a CSV File

The `rapidcsv`[6] header-only library

---

6. Copyright © 2017, Kristofer Berggren. All rights reserved.

```
https://github.com/d99kris/rapidcsv
```

provides class `rapidcsv::Document` that you can use to read and manipulate CSV files.[7] Many other libraries have built-in CSV support. For your convenience, we provided `rapidcsv` in the `examples` folder's `libraries/rapidcsv` subfolder. As in earlier examples that use open-source libraries, you'll need to point the compiler at the `rapidcsv` subfolder's `src` folder so you can include `<rapidcsv.h>` (Fig. 8.14, line 5).

---

7. Another popular data format is JavaScript Object Notation (JSON). There are C++ libraries for reading and generating JSON, such as RapidJSON (https://github.com/Tencent/rapidjson) and cereal (https://uscilab.github.io/cereal/index.html; discussed in Section 9.22).

```
 1  // fig08_14.cpp
 2  // Reading from a CSV file.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5  #include <rapidcsv.h>
 6  #include <vector>
 7
 8  int main() {
 9     rapidcsv::Document document{"accounts.csv"}; // loads accounts.csv
10     std::vector<int> accounts{document.GetColumn<int>("account")};
11     std::vector<std::string> names{
12        document.GetColumn<std::string>("name")};
13     std::vector<double> balances{document.GetColumn<double>("balance")};
14
15    std::cout << fmt::format(
16        "{:<10}{:<13}{:>7}\n", "Account", "Name", "Balance");
17
18     for (size_t i{0}; i < accounts.size(); ++i) {
19        std::cout << fmt::format("{:<10}{:<13}{:>7.2f}\n",
20           accounts.at(i), names.at(i), balances.at(i));
21     }
22  }
```

```
Account    Name         Balance
```

```
100         Jones          24.98
200         Doe           345.67
300         White           0.00
400         Stone         -42.16
500         Rich          224.62
```

Fig. 8.14 Reading from a CSV file.

Line 9 creates and initializes a `rapidcsv::Document` object named `document`. This statement loads the specified file (`"accounts.csv"`).[8] Class `Document`'s member functions enable you to work with the CSV data by row, by column or by individual value in a specific row and column. In this example, lines 10–13 get the data using the class's `GetColumn` template member function. This function returns the specified column's data as a `std::vector` containing elements of the type you specify in angle brackets. Line 10's call

8. The file `accounts.csv` must be in the same folder as this program's executable.

```
document.GetColumn<int>("account")
```

returns a `vector<int>` containing the account numbers for every record. Similarly, the calls in lines 11–12 and 13 return a `vector<string>` and a `vector<double>` containing all the records' names and balances, respectively. Lines 15–21 format and display the file's contents to confirm that they were read properly.

Caution: Commas in CSV Data Fields

Be careful when working with strings containing embedded commas, such as the name `"Jones, Sue"`. If this name were accidentally stored as the two strings `"Jones"` and `"Sue"`, that CSV record would have *four* fields, not *three*. Programs that read CSV files typically expect every record to have the same number of fields; otherwise, problems occur.

Caution: Missing Commas and Extra Commas in CSV Files

Be careful when preparing and processing CSV files. For example, suppose your file is composed of records, each with *four* comma-separated `int` values, such as

```
100,85,77,9
```

If you accidentally omit one of these commas, as in

```
100,8577,9
```

then the record has only *three* fields, one with the invalid value `8577`.
If you put two adjacent commas where only one is expected, as in

```
100,85,,77,9
```

then you have *five* fields rather than *four*, and one of the fields erroneously would be *empty*. Each of these comma-related errors could confuse programs trying to process the record.

8.19.2 Reading and Analyzing the `Titanic` *Disaster Dataset*

One of the most commonly used datasets for data analytics and data science beginners is the **Titanic disaster dataset**.[9] It lists all the passengers and whether they survived when the ship *Titanic* struck an iceberg and sank during its maiden voyage of April 10–15, 1912. We'll load the dataset in Fig. 8.15, view some of its data and perform some basic data analytics.

---

9. "*Titanic*" dataset on OpenML.org (https://www.openml.org/d/40945). Author: Frank E. Harrell, Jr., and Thomas Cason. Source: Vanderbilt Biostatistics (https://biostat.app.vumc.org/wiki/pub/Main/DataSets/titanic.html). The OpenML license terms (https://www.openml.org/cite) say, "You are free to use OpenML and all empirical data and metadata under the CC-BY license (https://creativecommons.org/licenses/by/4.0/), requesting appropriate credit if you do."

To download the dataset in CSV format, go to

```
https://www.openml.org/d/40945
```

and click the CSV download button in the page's upper-right corner. This downloads the file `phpMYEkMl.csv`, which we renamed as `titanic.csv`.

Getting to Know the Data

Much of data analytics and data science is devoted to getting to know your data. One way is simply to look at the raw data. If you open the `titanic.csv` file in a text editor or spreadsheet application, you'll see that the dataset contains 1,309 rows, each containing 14 columns—often called **features** in data analytics. We'll use only four columns here:

- `survived`: `1` or `0` for yes or no, respectively.
- `sex`: `"female"` or `"male"`.
- `age`: The passenger's age. Most ages are integers, but some children under 1 year of age have floating-point values, so we'll process this column as `double` values.
- `pclass`: 1, 2 or 3 for first class, second class or third class, respectively.

To learn more about the dataset's origins and its other columns, visit

```
https://biostat.app.vumc.org/wiki/pub/Main/DataSets/titanic3info.txt
```

Missing Data

Bad data values and missing values can significantly impact data analysis. The *Titanic* dataset is missing `age`s for 263 passengers. These are represented as `?` in the CSV file. In this example, when we produce descriptive statistics for the passengers' ages, we'll filter out and ignore the missing values. Some data scientists advise against any attempts to insert "reasonable values." Instead, they advocate clearly marking missing data and leaving it up to a data analytics package to handle the issue. Others offer strong cautions.[10]

10. This footnote was abstracted from a comment sent to us July 20, 2018, by one of our Python textbook's academic reviewers, Dr. Alison Sanchez of the University of San Diego School of Business. She commented: "Be cautious when mentioning 'substituting reasonable values' for missing or bad

values. A stern warning: ‘Substituting’ values that increase statistical significance or give more ‘reasonable’ or ‘better’ results is not permitted. ‘Substituting’ data should not turn into ‘fudging’ data. The first rule students should learn is not to eliminate or change values that contradict their hypotheses. ‘Substituting reasonable values‘ does not mean students should feel free to change values to get the results they want.”

Loading the Dataset

20 Figure 8.15 uses some C++20 `ranges` library features we introduced in Section 6.14. We broke the program into parts for discussion purposes. Lines 15–17 create and initialize a `rapidcsv::Document` object named `titanic` that loads `"titanic.csv"`.[11] The second and third arguments in line 16 are two of the default arguments used to initialize a `Document` object when you create it only by specifying the CSV filename. Recall from our discussion of default function arguments that when an argument is specified explicitly for a given parameter, all prior arguments in the argument list also must be specified explicitly. We provided the second and third arguments only so we could specify the fourth argument.

11. The file `titanic.csv` must be in the same folder as this program’s executable.

```
1   // fig08_15.cpp
2   // Reading the Titanic dataset from a CSV file, then analyzing it.
3   #include <fmt/format.h>
4   #include <algorithm>
5   #include <cmath>
6   #include <iostream>
7   #include <numeric>
8   #include <ranges>
9   #include <rapidcsv.h>
10  #include <string>
11  #include <vector>
12
13  int main() {
14     // load Titanic dataset; treat missing age values as NaN
15     rapidcsv::Document titanic{"titanic.csv",
16        rapidcsv::LabelParams{}, rapidcsv::SeparatorParams{},
17        rapidcsv::ConverterParams{true}};
18
```

Fig. 8.15 Reading the *Titanic* dataset from a CSV file, then analyzing it.

The `rapidcsv::LabelParams{}` argument specifies by default that the CSV file’s first row contains the column names. The `rapidcsv::SeparatorParams{}` argument specifies by default that each record’s fields are separated by commas. The fourth argument

```
rapidcsv::ConverterParams{true}
```

enables RapidCSV to convert missing and bad data values in integer columns to 0 and floating-point columns to NaN (not a number). This enables us to load all the data in the `age` column into a `vector<double>`, including the missing values represented by `?`.

Loading the Data to Analyze

Lines 20–23 use the `rapidcsv::Document`'s `GetColumn` member function to get each column by name.

```
19     // GetColumn returns column's data as a vector of the appropriate type
20     auto survived{titanic.GetColumn<int>("survived")};
21     auto sex{titanic.GetColumn<std::string>("sex")};
22     auto age{titanic.GetColumn<double>("age")};
23     auto pclass{titanic.GetColumn<int>("pclass")};
```

Viewing Some Rows in the Titanic Dataset

The 1,309 rows each represent one passenger. According to Wikipedia, there were approximately 1,317 passengers and 815 of them died.[12] For large datasets, it's not possible to display all the data at once. A common practice when getting to know your data is to display a few rows from the beginning and end of the dataset, so you can get a sense of the data. The code in lines 26–31 displays the first five elements of each column's data:

12. "Passengers of the *Titanic.*" Wikipedia. Wikimedia Foundation. Accessed January 3, 2022. https://en.wikipedia.org/wiki/Passengers_of_the_RMS_Titanic.

```
25     // display first 5 rows
26     std::cout << fmt::format("First five rows:\n{:<10}{:<8}{:<6}{}\n",
27        "survived", "sex", "age", "class");
28     for (size_t i{0}; i < 5; ++i) {
29        std::cout << fmt::format("{:<10}{:<8}{:<6.1f}{}\n",
30           survived.at(i), sex.at(i), age.at(i), pclass.at(i));
31     }
```

```
First five rows:
survived  sex      age   class
1         female   29.0  1
1         male     0.9   1
0         female   2.0   1
0         male     30.0  1
0         female   25.0  1
```

The code in lines 34–40 displays the last five elements of each column's data. To determine the control variable's starting value, line 36 calls the `rapidcsv::Document`'s `GetRowCount` member function. Then line 37 initializes the control variable to five less than the row count. Note the value `nan`, indicating a missing value for one of the `age` column's rows.

```
33     // display last 5 rows
34     std::cout << fmt::format("\nLast five rows:\n{:<10}{:<8}{:<6}{}\n",
35        "survived", "sex", "age", "class");
36     const auto count{titanic.GetRowCount()};
37     for (size_t i{count - 5}; i < count; ++i) {
38        std::cout << fmt::format("{:<10}{:<8}{:<6.1f}{}\n",
39           survived.at(i), sex.at(i), age.at(i), pclass.at(i));
40     }
41
```

```
Last five rows:
```

```
survived  sex     age   class
0         female  14.5  3
0         female  nan   3
0         male    26.5  3
0         male    27.0  3
0         male    29.0  3
```

Basic Descriptive Statistics

20 As part of getting to know a dataset, data scientists often use statistics to describe and summarize data. Let's calculate several **descriptive statistics** for the age column, including the number of passengers for which we have age values, and the average, minimum, maximum and median age values. Before performing these calculations, we must remove `nan` values. A calculation that includes the value `nan` produces `nan` as the calculation's result. Lines 43–44 use the C++20 ranges filtering techniques from Section 6.14 to keep only the values in the `age vector` that are *not* `nan`. Function `isnan` (header `<cmath>`) returns `true` if the value is `nan`. Next, lines 45–46 create a `vector<double>` named `cleanAge`. The `vector` initializes its elements by iterating through the filtered results in the `removeNaN` pipeline.

```
42     // use C++20 ranges to eliminate missing values from age column
43     auto removeNaN{
44        age | std::views::filter([](const auto& x) {return !isnan(x);})};
45     std::vector<double> cleanAge{
46        std::begin(removeNaN), std::end(removeNaN)};
```

Basic Descriptive Statistics for the Cleaned Age Column

Now, we can calculate the descriptive statistics. Line 49 sorts `cleanAge`, which will help us determine the minimum, maximum and median values. To count the number of people for which we have valid `age`s, we simply get `cleanAge`'s `size` (line 50).

```
48     // descriptive statistics for cleaned ages column
49     std::sort(std::begin(cleanAge), std::end(cleanAge));
50     size_t size{cleanAge.size()};
51     double median{};
52
53     if (size % 2 == 0) { // find median value for even number of items
54        median = (cleanAge.at(size / 2 - 1) + cleanAge.at(size / 2)) / 2;
55     }
56     else { // find median value for odd number of items
57        median = cleanAge.at(size / 2);
58     }
59
60     std::cout << "\nDescriptive statistics for the age column:\n"
61        << fmt::format("Passengers with age data: {}\n", size)
62        << fmt::format("Average age: {:.2f}\n", std::accumulate(
63           std::begin(cleanAge), std::end(cleanAge), 0.0) / size)
64        << fmt::format("Minimum age: {:.2f}\n", cleanAge.front())
65        << fmt::format("Maximum age: {:.2f}\n", cleanAge.back())
66        << fmt::format("Median age: {:.2f}\n", median);
67
```

```
Descriptive statistics for the age column:
Passengers with age data: 1046
Average age: 29.88
Minimum age: 0.17
Maximum age: 80.00
Median age: 28.00
```

Lines 51–58 determine the median. If `cleanAge`'s `size` is even, the median is the average of the two middle elements (line 54); otherwise, it's the middle element (line 57). Lines 60–66 display the descriptive statistics. Lines 62–63 calculate the average age by using the `accumulate` algorithm to total the ages, then dividing the result by `size`. The `vector` is sorted, so lines 64–65 determine the minimum and maximum values by calling the `vector`'s **`front`** and **`back`** member functions to get the `vector`'s first and last elements, respectively. The average and median are **measures of central tendency**. Each is a way to produce a single value that represents a "central" value in a set of values—that is, a value which is, in some sense, typical of the others.

For the `1046` people with valid `age`s, the average `age` was `29.88` years old. The youngest passenger (i.e., the minimum) was just over two months old (`0.17 * 12` is `2.04`), and the oldest (i.e., the maximum) was 80. The `median age` was `28`.

Determining Passenger Counts By Class

20 Let's calculate each class's number of passengers. Lines 69–73 define a lambda that counts the number of passengers in a particular class. C++20's **`std::ranges::count_if` algorithm** counts all the elements in its first argument for which the lambda in its second argument returns `true`. A benefit of the `std::ranges` algorithms is that you do not need to specify the beginning and end of the container—the `std::ranges` algorithms handle this for you, simplifying your code. The first argument to `count_if` in this example will be the `vector<int>` named `pclass`. In line 72's lambda

```
[classNumber](int x) {return classNumber == x;}
```

the **lambda introducer** `[classNumber]` specifies that the `countClass` lambda's parameter `classNumber` is used in this lambda's body—this is known as **capturing** the `countClass` variable. By default, capturing is done by value, so line 72's lambda captures a copy of `classNumber`'s value. We'll say more about capturing lambdas and the `std::ranges` algorithms in Chapter 14. Lines 75–77 define constants for the three passenger classes. Lines 78–80 call the `countClass` lambda for each passenger class, and lines 82–84 display the counts.

```
68     // passenger counts by class
69     auto countClass{
70        [](const auto& column, const int classNumber) {
71           return std::ranges::count_if(column,
72              [classNumber](int x) {return classNumber == x;});
73     };
74
75     constexpr int firstClass{1};
76     constexpr int secondClass{2};
77     constexpr int thirdClass{3};
78     const auto firstCount{countClass(pclass, firstClass)};
79     const auto secondCount{countClass(pclass, secondClass)};
80     const auto thirdCount{countClass(pclass, thirdClass)};
```

```
81
82     std::cout << "\nPassenger counts by class:\n"
83        << fmt::format("1st: {}\n2nd: {}\n3rd: {}\n\n",
84           firstCount, secondCount, thirdCount);
85
```

```
Passenger counts by class:
1st: 323
2nd: 277
3rd: 709
```

Basic Descriptive Statistics for the Cleaned Age Column

20 Let's say you want to determine some statistics about people who survived. Lines 87–89 define a lambda that counts survivors using C++20's `std::ranges::count_if` algorithm counts. Recall that the `survived` column contains `1` or `0` to represent survived or died, respectively. These also are values that C++ can treat as `true` (`1`) or `false` (`0`), so the lambda in line 88

```
[](auto x) {return x;}
```

simply returns the column value. If that value is `1`, `count_if` counts that element. To determine how many people died, line 92 simply subtracts `survivorCount` from the `survived vector`'s size. Line 94 calculates the percentage of people who survived.

```
86     // percentage of people who survived
87     const auto survivorCount{
88        std::ranges::count_if(survived, [](auto x) {return x;})
89     };
90
91     std::cout << fmt::format("Survived count: {}\nDied count: {}\n",
92                survivorCount, survived.size() - survivorCount);
93     std::cout << fmt::format("Percent who survived: {:.2f}%\n\n",
94                100.0 * survivorCount / survived.size());
95
```

```
Survived count: 500
Died count: 809
Percent who survived: 38.20%
```

Counting By Sex and By Passenger Class the Numbers of People Who Survived

Lines 97–117 iterate through the `survived` column, using its `1` or `0` value as a condition (line 104). For each survivor, we increment counters for the survivor's `sex` (`surviving-Women` and `survivingMen` in line 105) and `pclass` (`surviving1st`, `surviving2nd` and `surviving3rd` in lines 107–115). We'll use these counts to calculate percentages.

```
96     // count who survived by male/female, 1st/2nd/3rd class
97     int survivingMen{0};
98     int survivingWomen{0};
99     int surviving1st{0};
100    int surviving2nd{0};
101    int surviving3rd{0};
```

```
102
103    for (size_t i{0}; i < survived.size(); ++i) {
104       if (survived.at(i)) {
105          sex.at(i) == "female" ? ++survivingWomen : ++survivingMen;
106
107          if (firstClass == pclass.at(i)) {
108             ++surviving1st;
109          }
110          else if (secondClass == pclass.at(i)) {
111             ++surviving2nd;
112          }
113          else { // third class
114             ++surviving3rd;
115          }
116       }
117    }
```

Calculating Percentages of People Who Survived

Lines 120–129 calculate and display the percentages of:

- women who survived,
- men who survived,
- first-class passengers who survived,
- second-class passengers who survived, and
- third-class passengers who survived.

Of the survivors, about two-thirds were women, and first-class passengers survived at a higher rate than the other passenger classes.

```
119    // percentages who survived by male/female, 1st/2nd/3rd class
120    std::cout << fmt::format("Female survivor percentage: {:.2f}%\n",
121                 100.0 * survivingWomen / survivorCount)
122          << fmt::format("Male survivor percentage: {:.2f}%\n\n",
123                100.0 * survivingMen / survivorCount)
124          << fmt::format("1st class survivor percentage: {:.2f}%\n",
125                100.0 * surviving1st / survivorCount)
126          << fmt::format("2nd class survivor percentage: {:.2f}%\n",
127                100.0 * surviving2nd / survivorCount)
128          << fmt::format("3rd class survivor percentage: {:.2f}%\n",
129                100.0 * surviving3rd / survivorCount);
130 }
```

```
Female survivor percentage: 67.80%
Male survivor percentage: 32.20%

1st class survivor percentage: 40.00%
2nd class survivor percentage: 23.80%
3rd class survivor percentage: 36.20%
```

8.20 Objects-Natural Case Study: Intro to Regular Expressions

Sometimes you'll need to recognize patterns in text, like phone numbers, e-mail addresses, ZIP codes, webpage addresses, Social Security numbers and more. A **regular expression** string describes a search pattern for matching characters in other strings. Regular expressions can help you extract data from unstructured text, such as social media posts. They're also important for ensuring that data is in the proper format before you attempt to process it.

Validating Data

Before working with text data, you'll often use regular expressions to validate it. For example, you might check that:

- A U.S. ZIP code consists of five digits (such as 02215) or five digits followed by a hyphen and four more digits (such as 02215-4775).
- A string last name contains only letters, spaces, apostrophes and hyphens.
- An e-mail address contains only the allowed characters in the allowed order.
- A U.S. Social Security number contains three digits, a hyphen, two digits, a hyphen and four digits, and adheres to other rules about the specific numbers used in each group of digits.

You'll rarely need to create your own regular expressions for common items like these. The following free websites

- https://regex101.com
- https://regexr.com/
- http://www.regexlib.com
- https://www.regular-expressions.info

and others offer repositories of existing regular expressions that you can copy and use. Many sites like these also allow you to test regular expressions to determine whether they'll meet your needs.

Other Uses of Regular Expressions

Regular expressions also are used to:

- Extract data from text (known as *scraping*)—for example, locating all URLs in a webpage.
- Clean data—for example, removing data that's not required, removing duplicate data, handling incomplete data, fixing typos, ensuring consistent data formats, removing formatting, changing text case, dealing with outliers and more.
- Transform data into other formats—for example, reformatting tab-separated or space-separated values into comma-separated values (CSV) for an application that requires data to be in CSV format. We discussed CSV in Section 8.19.

Supported Regular Expression "Flavors"

C++ supports various regular-expression grammars—often called "flavors." C++ uses a slightly modified version of ECMAScript regular expressions by default.[13] For the complete list of supported grammars, see

13. "Modified ECMAScript Regular Expression Grammar." Accessed January 4, 2022. https://en.cppreference.com/w/cpp/regex/ecmascript.

```
https://en.cppreference.com/w/cpp/regex/syntax_option_type
```

To use regular expressions, include the header `<regex>`, which provides several classes and functions for recognizing and manipulating regular expressions.[14] Figure 8.16 demonstrates matching entire strings to regular expression patterns. We broke the program into parts for discussion purposes.

14. Some C++ programmers prefer to use third party regular-expression libraries, such as RE2 or PCRE. For a list of other C++ regular expression libraries, see https://github.com/fffaraz/awesome-cpp#regular-expression.

```
1  // fig08_16.cpp
2  // Matching entire strings to regular expressions.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <regex>
6
7  int main() {
8     // fully match a pattern of literal characters
9     std::regex r1{"02215"};
10    std::cout << "Matching against: 02215\n"
11        << fmt::format("02215: {}; 51220: {}\n\n",
12              std::regex_match("02215", r1), std::regex_match("51220", r1));
13
```

```
Matching against: 02215
02215: true; 51220: false
```

Fig. 8.16 Matching entire strings to regular expressions.

Matching Literal Characters

The `<regex>` function `regex_match` returns `true` if the entire string in its first argument matches the pattern in its second argument. By default, pattern matching is case sensitive—later, you'll see how to perform case-insensitive matches. Let's begin by matching literal characters—that is, characters that match themselves. Line 9 creates a `regex` object `r1` for the pattern `"02215"`, containing only literal digits that match themselves in the specified order. Line 12 calls `regex_match` for the strings `"02215"` and `"51220"`. Each has the same digits, but **only `"02215"` has them in the correct order for a match**.

Metacharacters, Character Classes and Quantifiers

Regular expressions typically contain various special symbols called **metacharacters**:

```
[] {} () \ * + ^ $ ? . |
```

The **`\` metacharacter** begins each of the predefined **character classes**, several of which are shown in the following table with the groups of characters they match.

Character class	Matches
`\d`	Any digit (0–9).
`\D`	Any character that is not a digit.

Character class	Matches
`\s`	Any whitespace character (such as spaces, tabs and newlines).
`\S`	Any character that is not a whitespace character.
`\w`	Any **word character** (also called an **alphanumeric character**)—that is, any uppercase or lowercase letter, any digit or an underscore
`\W`	Any character that is not a word character.

To match any metacharacter as its literal value, precede it by a backslash. For example, `\$` matches a dollar sign (`$`) and `\\` matches a backslash (`\`).

Matching Digits

Let's validate a five-digit ZIP code. In the regular expression `\d{5}` (created by the raw string literal in line 15), `\d` is a character class representing a digit (0–9). A character class is a **regular expression escape sequence** that **matches one character**. To match more than one, follow the character class with a **quantifier**. The quantifier `{5}` repeats `\d` five times, as if we had written `\d\d\d\d\d`, to match five consecutive digits. Line 18's second call to `regex_match` returns `false` because `"9876"` contains only four consecutive digits.

```
14     // fully match five digits
15     std::regex r2{R"(\d{5})"};
16     std::cout << R"(Matching against: \d{5})" << "\n"
17        << fmt::format("02215: {}; 9876: {}\n\n",
18              std::regex_match("02215", r2), std::regex_match("9876", r2));
19
```

```
Matching against: \d{5}
02215: true; 9876: false
```

Custom Character Classes

Characters in square brackets, `[]`, define a **custom character class** that **matches a single character**. For example, `[aeiou]` matches a lowercase vowel, `[A-Z]` matches an uppercase letter, `[a-z]` matches a lowercase letter and `[a-zA-Z]` matches any lowercase or uppercase letter. Line 21 defines a custom character class to validate a simple first name with no spaces or punctuation.

```
20     // match a word that starts with a capital letter
21     std::regex r3{"[A-Z][a-z]*"};
22     std::cout << "Matching against: [A-Z][a-z]*\n"
23        << fmt::format("Wally: {}; eva: {}\n\n",
24              std::regex_match("Wally", r3), std::regex_match("eva", r3));
```

```
Matching against: [A-Z][a-z]*
```

```
Wally: true; eva: false
```

A first name might contain many letters. In the regular expression `r3` (line 21), `[A-Z]` matches one uppercase letter, and `[a-z]*` matches any number of lowercase letters. The **`*` quantifier matches zero or more occurrences of the subexpression to its left** (in this case, `[a-z]`). So `[A-Z][a-z]*` matches `"Amanda"`, `"Bo"` or even `"E"`.

When a custom character class starts with a **caret (`^`)**, the class **matches any character that's not specified**. So `[^a-z]` (line 27) matches any character that's not a lowercase letter.

```
26     // match any character that's not a lowercase letter
27     std::regex r4{"[^a-z]"};
28     std::cout << "Matching against: [^a-z]\n"
29        << fmt::format("A: {}; a: {}\n\n",
30              std::regex_match("A", r4), std::regex_match("a", r4));
31
```

```
Matching against: [^a-z]
A: true; a: false
```

Metacharacters in a custom character class are treated as literal characters—that is, the characters themselves. So `[*+$]` (line 33) matches a single `*`, `+` or `$` character.

```
32     // match metacharacters as literals in a custom character class
33     std::regex r5{"[*+$]"};
34     std::cout << "Matching against: [*+$]\n"
35        << fmt::format("*: {}; !: {}\n\n",
36              std::regex_match("*", r5), std::regex_match("!", r5));
37
```

```
Matching against: [*+$]
*: true; !: false
```

* vs. + Quantifier

To require at least one lowercase letter in a first name, replace the `*` quantifier in line 21 with **`+`** (line 39). This **matches at least one occurrence of a subexpression to its left.**

```
38     // matching a capital letter followed by at least one lowercase letter
39     std::regex r6{"[A-Z][a-z]+"};
40     std::cout << "Matching against: [A-Z][a-z]+\n"
41        << fmt::format("Wally: {}; E: {}\n\n",
42              std::regex_match("Wally", r6), std::regex_match("E", r6));
43
```

```
Matching against: [A-Z][a-z]+
Wally: true; E: false
```

Both `*` and `+` are **greedy**—**they match as many characters as possible**. So the regular expression `[A-Z][a-z]+` matches `"Al"`, `"Eva"`, `"Samantha"`, `"Benjamin"` and any other words that begin with a capital letter followed by at least one lowercase letter. You can make `*` and `+` **lazy, so they match as few characters as possible** by appending a question mark (`?`) to the quantifier, as in `*?` or `+?`.

Other Quantifiers

The **`?` quantifier** by itself **matches zero or one occurrence of the subexpression to its left.** In the regular expression `label1?ed` (line 45), the subexpression is the literal character `"1"`. So in

the `regex_match` calls in lines 48–49, the regular expression matches `labelled` (the U.K. English spelling) and `labeled` (the U.S. English spelling), but in line 50's `regex_-match` call, the regular expression does not match the misspelled word `labellled`.

```
44     // matching zero or one occurrenctes of a subexpression
45     std::regex r7{"labell?ed"};
46     std::cout << "Matching against: labell?ed\n"
47        << fmt::format("labelled: {}; labeled: {}; labellled: {}\n\n",
48              std::regex_match("labelled", r7),
49              std::regex_match("labeled", r7),
50              std::regex_match("labellled", r7));
51
```

```
Matching against: labell?ed
labelled: true; labeled: true; labellled: false
```

You can use the **{*n*,} quantifier** to **match at least n occurrences of a subexpression to its left**. The regular expression in line 53 matches strings containing at least three digits.

```
52     // matching n (3) or more occurrences of a subexpression
53     std::regex r8{R"(\d{3,})"};
54     std::cout << R"(Matching against: \d{3,})" << "\n"
55        << fmt::format("123: {}; 1234567890: {}; 12: {}\n\n",
56              std::regex_match("123", r8),
57              std::regex_match("1234567890", r8),
58              std::regex_match("12", r8));
59
```

```
Matching against: \d{3,}
123: true; 1234567890: true; 12: false
```

You can use the **{*n*,*m*} quantifier** to **match between *n* and *m* (inclusive) occurrences** of a subexpression. The regular expression in line 61 matches strings containing 3 to 6 digits.

```
60     // matching n to m inclusive (3-6), occurrences of a subexpression
61     std::regex r9{R"(\d{3,6})"};
62     std::cout << R"(Matching against: \d{3,6})" << "\n"
63        << fmt::format("123: {}; 123456: {}; 1234567: {}; 12: {}\n",
64              std::regex_match("123", r9), std::regex_match("123456", r9),
65              std::regex_match("1234567", r9), std::regex_match("12", r9));
66  }
```

```
Matching against: \d{3,6}
123: true; 123456: true; 1234567: false; 12: false
```

8.20.2 Replacing Substrings

The header `<regex>` provides function **`regex_replace`** to replace patterns in a string. Let's convert a tab-delimited string to comma-delimited (Fig. 8.17). The `regex_replace` (line 13) function receives three arguments:

- the `string` to be searched (`"1\t2\t3\t4"`),

- the `regex` pattern to match (the tab character `"\t"`) and
- the replacement text (`","`).

```
// fig08_17.cpp
// Regular expression replacements.
#include <fmt/format.h>
#include <iostream>
#include <regex>
#include <string>

int main() {
   // replace tabs with commas
   std::string s1{"1\t2\t3\t4"};
   std::cout << fmt::format("Original string: {}\n", R"(1\t2\t3\t4)")
      << fmt::format("After replacing tabs with commas: {}\n",
            std::regex_replace(s1, std::regex{"\t"}, ","));
}
```

```
Original string: 1\t2\t3\t4
After replacing tabs with commas: 1,2,3,4
```

Fig. 8.17 Regular expression replacements.

It returns a new `string` containing the modifications. The expression

```
std::regex{"\t"}
```

creates a temporary `regex` object, initializes it and immediately passes it into function `regex_replace`. This is useful if you do not need to reuse a `regex` multiple times.

8.20.3 Searching for Matches

You can match a substring within a string using function **`regex_search`** (Fig. 8.18), which returns `true` if any part of an arbitrary `string` matches the regular expression. Optionally, the function also gives you access to the matching substring via an object of the class template **`match_results`** that you pass as an argument. There are `match_results` aliases for different string types:

- For searches in `std::string`s, use an **`smatch`** (pronounced “ess match”).
- For searches on C-strings and string literals, use a **`cmatch`** (pronounced “see match”).

```
// fig08_18.cpp
// Matching patterns throughout a string.
#include <fmt/format.h>
#include <iostream>
#include <regex>
#include <string>

int main() {
   // performing a simple match
   std::string s1{"Programming is fun"};
   std::cout << fmt::format("s1: {}\n\n", s1);
   std::cout << "Search anywhere in s1:\n"
      << fmt::format("Programming: {}; fun: {}; fn: {}\n\n",
```

```
14              std::regex_search(s1, std::regex{"Programming"}),
15              std::regex_search(s1, std::regex{"fun"}),
16              std::regex_search(s1, std::regex{"fn"}));
17
```

```
s1: Programming is fun

Search anywhere in s1:
Programming: true; fun: true; fn: false
```

Fig. 8.18 Matching patterns throughout a string.

The `<regex>` header has not yet been updated for `string_view`s.

Finding a Match Anywhere in a String

The calls to function `regex_search` in lines 14–16 search in `s1` (`"Programming is fun"`) for the first occurrence of a substring that matches a regular expression—in this case, the literal strings `"Programming"`, `"fun"` and `"fn"`. Function `regex_search`'s two-argument version simply returns `true` or `false` to indicate a match.

Ignoring Case in a Regular Expression and Viewing the Matching Text

The **`regex_constants`** in the header `<regex>` can customize how regular expressions perform matches. For example, matches are case sensitive by default, but by using the constant **`regex_constants::icase`**, you can perform a case-insensitive search.

```
18      // ignoring case
19      std::string s2{"SAM WHITE"};
20      std::smatch match; // store the text that matches the pattern
21      std::cout << fmt::format("s2: {}\n\n", s2);
22      std::cout << "Case insensitive search for Sam in s2:\n"
23         << fmt::format("Sam: {}\n", std::regex_search(s2, match,
24               std::regex{"Sam", std::regex_constants::icase}))
25         << fmt::format("Matched text: {}\n\n", match.str());
26
```

```
s2: SAM WHITE

Case insensitive search for Sam in s2:
Sam: true
Matched text: SAM
```

Lines 23–24 call `regex_search`'s three-argument version, in which the arguments are

- the string to search (`s2`; line 23), which in this case contains all capital letters,
- the `smatch` object (line 23), which stores the match if there is one, and
- the `regex` pattern to match (line 24).

Here, we created the `regex` with a second argument

```
std::regex{"Sam", std::regex_constants::icase}
```

This `regex` matches the literal characters `"Sam"` regardless of their case. So, in `s2`, `"SAM"` matches the `regex "Sam"` because both have the same letters, even though `"SAM"` contains only uppercase letters.

To confirm the matching text, line 25 gets the `match`'s `string` representation by calling its member function `str`.

Finding All Matches in a String

Let's extract all the U.S. phone numbers of the form `###-###-####` from a string. The following code finds each substring in `contact` (lines 28–29) that matches the `regex phone` (line 30) and displays the matching text.

```
27      // finding all matches
28      std::string contact{
29         "Wally White, Home: 555-555-1234, Work: 555-555-4321"};
30      std::regex phone{R"(\d{3}-\d{3}-\d{4})"};
31
32      std::cout <<fmt::format("Finding phone numbers in:\n{}\n", contact);
33      while (std::regex_search(contact, match, phone)) {
34         std::cout << fmt::format("   {}\n", match.str());
35         contact = match.suffix();
36      }
37   }
```

```
Finding phone numbers in:
Wally White, Home: 555-555-1234, Work: 555-555-4321
   555-555-1234
   555-555-4321
```

The `while` statement iterates as long as `regex_search` returns `true`—that is, as long as it finds a match. Each iteration of the loop

- displays the substring that matched the regular expression (line 34), then
- replaces `contact` with the result of calling the `match` object's member function `suffix` (line 35), which returns the portion of the string that has not yet been searched. This new `contact` string is used in the next call to `regex_search`.

8.21 Wrap-Up

This chapter discussed more details of `std::string`, such as assigning, concatenating, comparing, searching and swapping strings. We also introduced several member functions to determine string characteristics; to find, replace and insert characters in a string; and to convert `string`s to pointer-based strings, and vice versa. We performed input from and output to strings in memory. We also introduced functions for converting numeric values to `string`s and for converting `string`s to numeric values.

We presented sequential text-file processing using features from the header `<fstream>` to manipulate persistent data, then demonstrated string stream processing. In our first of two Objects-Natural case studies, we used an open-source library to read the contents of the *Titanic* dataset from a CSV file, then performed some basic data analytics. Our second Objects-Natural case study introduced regular expressions for pattern matching.

We've now introduced the basic concepts of control statements, functions, `array`s, `vector`s, `string`s and files. You've already seen in many of our Objects-Natural case studies that C++ applications typically create and manipulate objects that perform the work of the application. In Chapter 9, you'll learn how to implement your own custom classes and use objects of those classes in applications. We'll begin discussing class design and related software-engineering concepts.

9. Custom Classes

Objectives

In this chapter, you'll:

- Define a custom class and use it to create objects.
- Implement a class's behaviors as member functions and attributes as data members.
- Access and manipulate `private` data members through `public` *get* and *set* functions to enforce data encapsulation.
- Use a constructor to initialize an object's data.
- Separate a class's interface from its implementation for reuse.
- Access class members via the dot (`.`) and arrow (`->`) operators.
- Use destructors to perform "termination housekeeping" on objects that go out of scope.
- Assign the data members of one object to those of another.
- Create objects composed of other objects.
- Use `friend` functions and declare `friend` classes.
- Access non-`static` class members via the `this` pointer.
- Use `static` data members and member functions.
- Use `struct`s to create aggregate types, and use C++20 designated initializers to initialize aggregate members.
- Do an Objects-Natural case study that serializes objects using JavaScript Object Notation (JSON) and the `cereal` library.

Outline

9.1 Introduction
9.2 Test-Driving an `Account` Object
9.3 `Account` Class with a Data Member and *Set* and *Get* Member Functions
9.3.1 Class Definition
9.3.2 Access Specifiers `private` and `public`
9.4 `Account` Class: Custom Constructors
9.5 Software Engineering with *Set* and *Get* Member Functions
9.6 `Account` Class with a Balance
9.7 `Time` Class Case Study: Separating Interface from Implementation
9.7.1 Interface of a Class
9.7.2 Separating the Interface from the Implementation
9.7.3 Class Definition
9.7.4 Member Functions
9.7.5 Including the Class Header in the Source-Code File
9.7.6 Scope Resolution Operator (`::`)
9.7.7 Member Function `setTime` and Throwing Exceptions
9.7.8 Member Functions `to24HourString` and `to12HourString`
9.7.9 Implicitly Inlining Member Functions
9.7.10 Member Functions vs. Global Functions
9.7.11 Using Class `Time`
9.7.12 Object Size
9.8 Compilation and Linking Process
9.9 Class Scope and Accessing Class Members

9.10 Access Functions and Utility Functions
9.11 `Time` Class Case Study: Constructors with Default Arguments
9.11.1 Class `Time`
9.11.2 Overloaded Constructors and C++11 Delegating Constructors
9.12 Destructors
9.13 When Constructors and Destructors Are Called
9.14 `Time` Class Case Study: A Subtle Trap—Returning a Reference or a Pointer to a `private` Data Member
9.15 Default Assignment Operator
9.16 `const` Objects and `const` Member Functions
9.17 Composition: Objects as Members of Classes
9.18 `friend` Functions and `friend` Classes
9.19 The `this` Pointer
9.19.1 Implicitly and Explicitly Using the `this` Pointer to Access an Object's Data Members
9.19.2 Using the `this` Pointer to Enable Cascaded Function Calls
9.20 `static` Class Members: Classwide Data and Member Functions
9.21 Aggregates in C++20
9.21.1 Initializing an Aggregate
9.21.2 C++20: Designated Initializers
9.22 Objects-Natural Case Study: Serialization with JSON
9.22.1 Serializing a `vector` of Objects Containing `public` Data
9.22.2 Serializing a `vector` of Objects Containing `private` Data
9.23 Wrap-Up

9.1 Introduction[1]

Section 1.4 presented a friendly introduction to object orientation, discussing classes, objects, data members (attributes) and member functions (behaviors). In our Objects-Natural case studies, you've created objects of existing classes and called their member functions to make the objects perform powerful tasks without having to know how these classes worked internally.

---

1. This chapter depends on the terminology and concepts introduced in Section 1.4, A Brief Refresher on Object Orientation.

This chapter begins our deeper treatment of object-oriented programming as we craft valuable custom classes. C++ is an **extensible programming language**—each class you create becomes a new type you can use to create objects. Some development teams in industry work on applications that contain hundreds, or even thousands, of custom classes.

9.2 Test-Driving an `Account` Object

We begin our introduction to custom classes with three examples that create objects of an `Account` class representing a simple bank account. First, let's look at the `main` program and output, so you can see an object of our initial `Account` class in action. To help you prepare for the larger programs you'll encounter later in this book and in industry, we define the `Account` class and `main` in

separate files—`main` in `AccountTest.cpp` (Fig. 9.1) and class `Account` in `Account.h`, which we'll show in Fig. 9.2.

```
1  // Fig. 9.1: AccountTest.cpp
2  // Creating and manipulating an Account object.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <string>
6  #include "Account.h"
7
8  int main() {
9     Account myAccount{}; // create Account object myAccount
10
11    // show that the initial value of myAccount's name is the empty string
12    std::cout << fmt::format("Initial account name: {}\n",
13       myAccount.getName());
14
15    // prompt for and read the name
16    std::cout << "Enter the account name: ";
17    std::string name{};
18    std::getline(std::cin, name); // read a line of text
19    myAccount.setName(name); // put name in the myAccount object
20
21    // display the name stored in object myAccount
22    std::cout << fmt::format("Updated account name: {}\n",
23       myAccount.getName());
24 }
```

```
Initial account name:
Enter the account name: Jane Green
Updated account name: Jane Green
```

Fig. 9.1 Creating and manipulating an `Account` object.

Instantiating an Object

Typically, you cannot call a class's member functions until you create an object of that class.[2] Line 9

2. In Section 9.20, you'll see that static member functions are an exception.

```
Account myAccount{}; // create Account object myAccount
```

creates an object called `myAccount`. The variable's type is `Account`—the class we'll define in Fig. 9.2.

Headers and Source-Code Files

When we declare `int` variables, the compiler knows what `int` is—it's a fundamental type built into C++. In line 9, however, the compiler does not know in advance what type `Account` is—it's a **user-defined type**.

When packaged properly, new classes can be reused by other programmers. It's customary to place a reusable class definition in a file known as a **header** with a `.h` filename extension.[3] You include that header wherever you need to use the class, as you've been doing with C++ standard library and third-party library classes throughout this book.

3. C++ standard library headers, like `<iostream>`, do not use the `.h` filename extension and some C++ programmers prefer the `.hpp` extension.

We tell the compiler what an `Account` is by including its header, as in line 6:

```
#include "Account.h"
```

If we omit this, the compiler issues error messages wherever we use class `Account` and any of its capabilities. A header that you define in your program is placed in double quotes (`""`), rather than the angle brackets (`<>`). The double quotes tell the compiler to check the folder containing `AccountTest.cpp` (Fig. 9.1) before the compiler's header search path.[4]

4. Even your custom class headers can be `#included` by placing them in angle brackets (< and >) by specifying your program's folder as part of the compiler's header search path, as you've done for third-party libraries in several Objects-Natural case studies.

Calling Class `Account`'s `getName` Member Function

Class `Account`'s `getName` member function returns the name stored in a particular `Account` object. Line 13 calls `myAccount.getName()` to get the `myAccount` object's initial name, which is the empty `string`. We'll say more about this shortly.

Calling Class `Account`'s `setName` Member Function

The `setName` member function stores a name in a particular `Account` object. Line 19 calls `myAccount`'s `setName` member function to store `name`'s value in the object `myAccount`.

Displaying the Name That Was Entered by the User

To confirm that `myAccount` now contains the name you entered, line 23 calls member function `getName` again and displays its result.

9.3 `Account` Class with a Data Member and *Set* and *Get* Member Functions

Now that we've seen class `Account` in action (Fig. 9.1), we present its internal details.

9.3.1 Class Definition

Class `Account` (Fig. 9.2) contains the data member `m_name` (line 19) to store the account holder's name. Each object of a class has its own copy of the class's data members.[5] In Section 9.6, we'll add a `balance` data member to keep track of the money in each `Account`. Class `Account` also contains member functions:

5. In Section 9.20, you'll see that `static` data members are an exception.

- `setName` (lines 10–12), which stores a name in an `Account`, and
- `getName` (lines 15–17), which retrieves a name from an `Account`.

```
1  // Fig. 9.2: Account.h
```

```
 2  // Account class with a data member and
 3  // member functions to set and get its value.
 4  #include <string>
 5  #include <string_view>
 6
 7  class Account {
 8  public:
 9     // member function that sets m_name in the object
10     void setName(std::string_view name) {
11        m_name = name; // replace m_name's value with name
12     }
13
14     // member function that retrieves the account name from the object
15     const std::string& getName() const {
16        return m_name; // return m_name's value to this function's caller
17     }
18  private:
19     std::string m_name; // data member containing account holder's name
20  }; // end class Account
```

Fig. 9.2 `Account` class with a data member and member functions to *set* and *get* its value.

Keyword `class` and the Class Body

The class definition begins with the keyword `class` (line 7), followed immediately by the class's name (`Account`). By convention:

- each word in a class name starts with a capital first letter and
- data-member and member-function names begin with a lowercase first letter, and each subsequent word begins with a capital first letter.

Every class's body is enclosed in braces `{}` (lines 7 and 20). The class definition terminates with a required semicolon (line 20).

Data Member `m_name` of Type `std::string`

Recall from Section 1.4 that an object has attributes. These are implemented as data members. Each object maintains its own copy of these throughout its lifetime—that is, while the object exists in memory. Usually, a class also contains member functions that manipulate the data members of particular objects of the class.

Data members are declared inside a class definition but outside the class's member functions. Line 19

```
std::string m_name; // data member containing account holder's name
```

declares a `string` data member called `m_name`. The `"m_"` prefix is a common naming convention to indicate that a variable represents a data member. If there are many `Account` objects, each has its own `m_name`. Because `m_name` is a data member, it can be manipulated by the class's member functions. Recall that the default `string` value is the empty `string` (`""`), which is why line 13 in `main` (Fig. 9.1) did not display a name.

By convention, C++ programmers typically place a class's data members last in the class's body. You can declare the data members anywhere in the class outside its member-function definitions, but scattering the data members can lead to hard-to-read code.

Use `std::` with Standard Library Components in Headers

Err Throughout the `Account.h` header (Fig. 9.2), we use `std::` when referring to `string_view` (line 10) and `string` (lines 15 and 19). Headers should not contain global scope `using` directives or `using` declarations. These would be `#include`d into other source files, potentially causing naming conflicts. Since Chapter 6, we've qualified every standard library class name, function name and object (e.g., `std::cout`) with `std::` as a good practice.

`setName` Member Function

SE Member function `setName` (lines 10–12) receives a `string_view` representing the `Account`'s name and assigns the `name` argument to data member `m_name`. Recall that a `string_view` is a read-only view into a character sequence, such as a `std::string` or a C-string. Line 11 copies parameter `name`'s characters into `m_name`. The `"m_"` in `m_name` makes it easy to distinguish the parameter `name` from the data member `m_name`.

`getName` Member Function

Member function `getName` (lines 15–17) has no parameters and returns a particular `Account` object's `m_name` to the caller as a `const std::string&`. Declaring the returned reference `const` ensures that the caller cannot modify the object's data via that reference.

`const` Member Functions

Note the `const` to the right of the parameter list in `getName`'s header (line 15). When returning `m_name`, member function `getName` does not, and should not, modify the `Account` object on which it's called. Declaring a member function `const` tells the compiler, "this function should not modify the object on which it's called—if it does, please issue a compilation error." This can help you locate errors if you accidentally insert code that would modify the object. It also tells the compiler that `getName` may be called on a `const Account` object or via a reference or pointer to a `const Account`.

9.3.2 Access Specifiers `private` and `public`

CG The keyword `private` (line 18) is an **access specifier**. Each access specifier is followed by a required colon (`:`). Data member `m_name`'s declaration (line 19) appears after `private:` to indicate that `m_name` is accessible only to class `Account`'s member functions.[6] This is known as **information hiding** (or hiding implementation details) and is a recommended practice of the C++ Core Guidelines[7] and object-oriented programming in general. The data member `m_name` is encapsulated (hidden) and can be used only in class `Account`'s `setName` and `getName` member functions. Most data-member declarations appear after the `private:` access specifier.[8]

6. Or to "friends" of the class, as you'll see in Section 9.18.

7. C++ Core Guidelines, “C.9: Minimize Exposure of Members.” Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-private.
8. We generally omit the colon when referring to private and `public` in sentences, as in this footnote.

This class also contains the `public` access specifier (line 8):

```
public:
```

SE Class members listed after access specifier `public`—and before the next access specifier if there is one—are “available to the public.” They can be used anywhere an object of the class is in scope. Making a class’s data members `private` facilitates debugging because problems with data manipulations are localized to the class’s member functions. We’ll say more about the benefits of `private` in Section 9.7. In Chapter 10, we’ll introduce the `protected` access specifier.

Default Access for Class Members

By default, class members are `private` unless you specify otherwise. After you list an access specifier, every subsequent class member has that access level until you list a different access specifier. The `public` and `private` access specifiers may be repeated, but this can be confusing. We prefer to list `public` only once, grouping all `public` members, and to list `private` only once, grouping all `private` members.

9.4 `Account` Class: Custom Constructors

As you saw in Section 9.3, when an `Account` object is created, its `string` data member `m_name` is initialized to the empty `string` by default. But what if you want to provide a name when you first create an `Account` object? Each class can define **constructors** that specify custom initialization for objects of that class. **A constructor is a special member function that must have the same name as the class**. Constructors cannot return values, so **they do not specify a return type** (not even `void`). C++ guarantees a constructor call when you create an object, so this is the ideal point to initialize an object’s data members. Every time you created an object so far in this book, the corresponding class’s constructor was called to initialize the object. As you’ll soon see, classes may have overloaded constructors.

Like member functions, constructors can have parameters—the corresponding argument values help initialize the object’s data members. For example, you can specify an `Account` object’s name when the object is created, as you’ll do in line 10 of Fig. 9.4:

```
Account account1{"Jane Green"};
```

Here, the `"Jane Green"` is passed to the `Account` class’s constructor to initialize the `account1` object’s data. The preceding statement assumes that class `Account` has a constructor that can receive a string argument.

`Account` Class Definition

Figure 9.3 shows class `Account` with a constructor that receives an `accountName` parameter and uses it to initialize the data member `m_name` when an `Account` object is created.

```
1  // Fig. 9.3: Account.h
2  // Account class with a constructor that initializes the account name.
3  #include <string>
4  #include <string_view>
5
6  class Account {
7  public:
8     // constructor initializes data member m_name with the parameter name
9     explicit Account(std::string_view name)
10       : m_name{name} { // member initializer
11       // empty body
12    }
13
14    // function to set the account name
15    void setName(std::string_view name) {
16       m_name = name;
17    }
18
19    // function to retrieve the account name
20    const std::string& getName() const {
21       return m_name;
22    }
23 private:
24    std::string m_name; // account name
25 }; // end class Account
```

Fig. 9.3 `Account` class with a constructor that initializes the account name.

`Account` Class's Custom Constructor Definition

Lines 9–12 of Fig. 9.3 define `Account`'s constructor. Usually, constructors are `public`, so any code with access to the class definition can create and initialize objects of that class. Line 9 indicates that the constructor has a `string_view` parameter. When creating a new `Account` object, you must pass a person's name to the constructor's `string_view` parameter. The constructor then initializes the data member `m_name` with the parameter's contents. Line 9 of Fig. 9.3 does not specify a return type (not even `void`) because constructors cannot return values. Also, constructors cannot be declared `const`—initializing an object must modify it.

The constructor's **member-initializer list** (line 10)

```
: m_name{name}
```

initializes the `m_name` data member. Member initializers appear between a constructor's parameter list and the left brace that begins the constructor's body. You separate the member initializer list from the parameter list with a colon (`:`).

SE Each member initializer consists of a data member's variable name followed by braces containing its initial value.[9] This member initializer calls the `std::string` class's constructor that receives a `string_view`. If a class has more than one data member, each member initializer is separated from the next by a comma. Member constructors execute in the order you declare the data members in the class. For clarity, list the member initializers in the same order. The member-initializer list executes before the constructor's body.

9. Occasionally, parentheses rather than braces may be required, such as when initializing a `vector` of a specified size, as we did in Fig. 6.14.

CG Perf Though you can perform initialization with assignment statements in the constructor's body, the C++ Core Guidelines recommend using member initializers.[10] You'll see later that member initializers can be more efficient. Also, you'll see that certain data members must be initialized using the member-initializer syntax because you cannot assign values to them in the constructor's body.

10. C++ Core Guidelines, "C.49: Prefer Initialization to Assignment in Constructors." Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-initialize.

`explicit` Keyword

CG Declare a constructor `explicit` if it can be called with one argument—that is, it has one parameter, or it has additional parameters with default arguments. This prevents the compiler from using the constructor to perform implicit type conversions.[11] The keyword `explicit` means that `Account`'s constructor must be called explicitly, as in

11. C++ Core Guidelines, "C.46: By Default, Declare Single-Argument Constructors `explicit`." Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-explicit.

```
Account account1{"Jane Green"};
```

Err For now, simply declare all single-parameter constructors `explicit`. In Section 11.9, you'll see that single-parameter constructors without `explicit` can be called implicitly to perform type conversions. Such implicit constructor calls can lead to subtle errors and are generally discouraged.

Initializing `Account` Objects When They're Created

The `AccountTest` program (Fig. 9.4) initializes two `Account` objects using the constructor. Line 10 creates the `Account` object `account1`:

```
Account account1{"Jane Green"};
```

```
1  // Fig. 9.4: AccountTest.cpp
2  // Using the Account constructor to initialize the m_name
3  // data member when each Account object is created.
4  #include <fmt/format.h>
5  #include <iostream>
6  #include "Account.h"
7
8  int main() {
9     // create two Account objects
10    Account account1{"Jane Green"};
11    Account account2{"John Blue"};
12
13    // display each Account's corresponding name
14    std::cout << fmt::format(
15       "account1 name is: {}\naccount2 name is: {}\n",
```

```
16          account1.getName(), account2.getName());
17    }
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 9.4 Using the `Account` constructor to initialize the `m_name` data member at the time each `Account` object is created.

This calls the `Account` constructor (lines 9–12 of Fig. 9.3), which uses the argument `"Jane Green"` to initialize the new object's `m_name` data member. Line 11 of Fig. 9.4 repeats this process, passing the argument `"John Blue"` to initialize `m_name` for `account2`:

```
Account account2{"John Blue"};
```

To confirm the objects were initialized properly, lines 14–16 call the `getName` member function on each `Account` object to get its name. The output shows that each `Account` has a different name, confirming that each object has its own copy of data member `m_name`.

Default Constructor

Recall that line 9 of Fig. 9.1 created an `Account` object with empty braces to the right of the object's variable name:

```
Account myAccount{};
```

In this statement, C++ implicitly calls `Account`'s **default constructor**. In any class that does not define a constructor, the compiler generates a default constructor with no parameters. The default constructor does not initialize the class's fundamental-type data members but does call the default constructor for each data member that's an object of another class. For example, though you do not see this in the first `Account` class's code (Fig. 9.2), `Account`'s default constructor calls class `std::string`'s default constructor to initialize the data member `m_name` to the empty string (`""`).

When you declare an object with empty braces as in the preceding statement, data members that are not explicitly initialized get **value initialized**—that is, they're zero-initialized, then the default constructor is called for each object. Without the braces, such data members contain undefined ("garbage") values.

There's No Default Constructor in a Class That Defines a Constructor

SE **If you define a custom constructor for a class, the compiler will not create a default constructor for that class.** In that case, you will not be able to create an `Account` object by calling the constructor with no arguments unless the custom constructor you define has an empty parameter list or has default arguments for all its parameters. We'll show later that you can force the compiler to create the default constructor even if you've defined non-default constructors. Unless default initialization of your class's data members is acceptable, initialize them in their declarations or provide a custom constructor that initializes them with meaningful values.

C++'s Special Member Functions

20 In addition to the default constructor, the compiler can generate default versions of five other **special member functions**—a copy constructor, a move constructor, a copy assignment operator, a move assignment operator and a destructor—as well as C++20's new three-way comparison operator. This chapter briefly introduces copy construction, copy assignment and

destructors. Chapter 11 introduces the three-way comparison operator and discusses the details of all these special member functions, including

- when you might need to define custom versions of each and
- the various C++ Core Guidelines for these special member functions.

You'll see that you should design your classes so the compiler can auto-generate the special member functions for you. This is called the "Rule of Zero," meaning that you provide no special member functions in the class's definition.

9.5 Software Engineering with `Set` and `Get` Member Functions

As you'll see in the next section, *set* and *get* member functions can validate attempts to modify `private` data and control how that data is presented to the caller, respectively. These are compelling software engineering benefits.

A **client** of the class is any other code that calls the class's member functions. If a data member were `public`, any client could see the data and do whatever it wanted with it, including setting it to an invalid value.

You might think that even though a client cannot directly access a `private` data member, the client can nevertheless do whatever it wants with the variable through `public` *set* and *get* functions. You'd think that you could peek at the `private` data (and see exactly how it's stored in the object) anytime with the `public` *get* function and that you could modify the `private` data at will through the `public` *set* function.

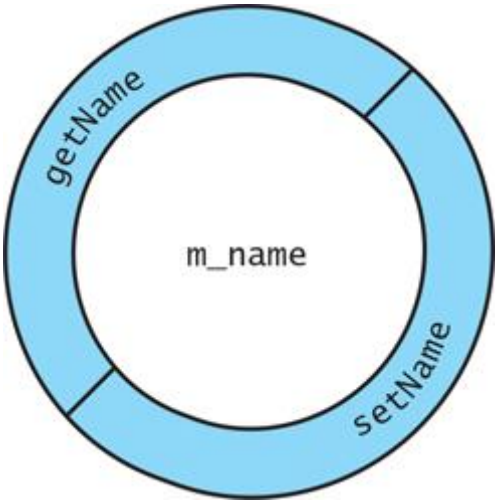
SE Actually, *set* functions can be written to validate their arguments and reject any attempts to *set* the data to incorrect values, such as

- a negative body temperature,
- a day in March outside the range 1 through 31 or
- a product code not in the company's product catalog.

SE SE A *get* function can present the data in a different form, keeping the object's actual data representation hidden from the user. For example, a `Grade` class might store a numeric grade as an `int` between 0 and 100, but a `getGrade` member function might return a letter grade as a `string`, such as `"A"` for grades between 90 and 100, `"B"` for grades between 80 and 89, etc. Tightly controlling the access to and presentation of `private` data can reduce errors while increasing your programs' robustness, security and usability.

Conceptual View of an `Account` Object with Encapsulated Data

You can think of an `Account` object as shown in the following diagram. The `private` data member `m_name`, represented by the inner circle, is hidden inside the object and accessible only via an outer layer of `public` member functions, represented by the outer ring containing `getName` and `setName`. Any client that needs to interact with the `Account` object can do so only by calling the `public` member functions of the outer layer.



SE Generally, data members are `private`, and the member functions that a class's clients need to use are `public`. Later, we'll discuss why you might use a `public` data member or a `private` member function. Using `public` *set* and *get* functions to control access to `private` data makes programs clearer and easier to maintain. Change is the rule rather than the exception. You should anticipate that your code will be modified—possibly often.

9.6 `Account` Class with a Balance

CG Figure 9.5 defines an `Account` class that maintains two related pieces of data—the account holder's name and bank balance. The C++ Core Guidelines recommend defining related data items in a class (or, as you'll see in Section 9.21, in a `struct`).[12]

12. C++ Core Guidelines, "C.1: Organize Related Data into Structures (`structs` or `classes`)." Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-org.

```
1  // Fig. 9.5: Account.h
2  // Account class with m_name and m_balance data members, and a
3  // constructor and deposit function that each perform validation.
4  #include <algorithm>
5  #include <string>
6  #include <string_view>
7
8  class Account {
9  public:
10    // Account constructor with two parameters
11    Account(std::string_view name, double balance)
12       : m_name{name}, m_balance{std::max(0.0, balance)} { // member init
13       // empty body
14    }
15
16    // function that deposits (adds) only a valid amount to the balance
17    void deposit(double amount) {
18       if (amount > 0.0) { // if the amount is valid
19          m_balance += amount; // add it to m_balance
20       }
21    }
22
23    // function that returns the account balance
```

```
24      double getBalance() const {
25         return m_balance;
26      }
27
28      // function that sets the account name
29      void setName(std::string_view name) {
30         m_name = name; // replace m_name's value with name
31      }
32
33      // function that returns the account name
34      const std::string& getName() const {
35         return m_name;
36      }
37   private:
38      std::string m_name;
39      double m_balance;
40   }; // end class Account
```

Fig. 9.5 `Account` class with `m_name` and `m_balance` data members, and a constructor and `deposit` function that each perform validation.

Data Member `balance`

A bank services many accounts, each with its own balance. Every `Account` object now has its own `m_name` and `m_balance`. Line 39 declares a `double` data member `m_balance`:[13]

13. As a reminder, industrial-strength financial applications should not use `double` to represent monetary amounts.

Two-Parameter Constructor

The class has a constructor and four member functions. It's common for someone opening an account to deposit money immediately, so the constructor (lines 11–14) receives a second parameter `balance` (a `double`) that represents the starting balance. We did not declare this constructor `explicit` because it cannot be called with only one parameter.

The `m_balance` member initializer calls the `std::max` function (header `<algorithm>`) to initialize `m_balance` to `0.0` or `balance`, whichever is greater. This ensures `m_balance` has a valid non-negative value when creating an `Account` object. Later, we'll validate arguments in the constructor body and use exceptions to indicate invalid arguments.

`deposit` Member Function

Member function `deposit` (lines 17–21) receives a `double` parameter `amount` and does not return a value. Lines 18–20 ensure that parameter `amount`'s value is added to `m_balance` only if `amount` is greater than zero (that is, it's a valid deposit amount).

`getBalance` Member Function

Member function `getBalance` (lines 24–26) allows the class's clients to obtain a particular `Account` object's `m_balance` value. The member function specifies return type `double` and an empty parameter list. Like `getName`, `getBalance` is declared `const` because returning `m_balance`'s value does not, and should not, modify the `Account` object on which it's called.

Manipulating `Account` Objects with Balances

The `main` function in Fig. 9.6 creates two `Account` objects (lines 8–9). It initializes them with a valid balance of `50.00` and an invalid balance of `-7.00`, respectively. Lines 12–15 output both `Account`s' names and balances by calling each object's `getName` and `getBalance` member functions. Our class ensures that

initial balances are greater than or equal to zero. The `account2` object's `balance` is initially `0.0` because the constructor rejected the attempt to start `account2` with a negative balance by setting its value to `0.0`.

```
1  // Fig. 9.6: AccountTest.cpp
2  // Displaying and updating Account balances.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include "Account.h"
6
7  int main() {
8     Account account1{"Jane Green", 50.00};
9     Account account2{"John Blue", -7.00};
10
11    // display initial balance of each object
12    std::cout << fmt::format("account1: {} balance is ${:.2f}\n",
13       account1.getName(), account1.getBalance());
14    std::cout << fmt::format("account2: {} balance is ${:.2f}\n\n",
15       account2.getName(), account2.getBalance());
16
account1: Jane Green balance is $50.00
account2: John Blue balance is $0.00
```

Fig. 9.6 Displaying and updating `Account` balances.

Reading a Deposit Amount from the User and Making a Deposit

Lines 17–21 prompt for, input and display the `account1` deposit amount. Line 22 calls object `account1`'s `deposit` member function, passing the `amount` to add to `account1`'s `balance`. Lines 25–28 output both `Account`s' names and balances again to show that only `account1`'s balance has changed.

```
17    std::cout << "Enter deposit amount for account1: "; // prompt
18    double amount;
19    std::cin >> amount; // obtain user input
20    std::cout << fmt::format(
21       "adding ${:.2f} to account1 balance\n\n", amount);
22    account1.deposit(amount); // add to account1's balance
23
24    // display balances
25    std::cout << fmt::format("account1: {} balance is ${:.2f}\n",
26       account1.getName(), account1.getBalance());
27    std::cout << fmt::format("account2: {} balance is ${:.2f}\n\n",
28       account2.getName(), account2.getBalance());
29
Enter deposit amount for account1: 25.37
adding $25.37 to account1 balance

account1: Jane Green balance is $75.37
account2: John Blue balance is $0.00
```

Lines 30–33 prompt for, input and display `account2`'s deposit amount. Line 34 calls object `account2`'s `deposit` member function with variable `amount` as the argument to add that value

to `account2`'s balance. Finally, lines 37–40 output both `Account`s' names and balances again to show that only `account2`'s balance has changed.

```
30      std::cout << "Enter deposit amount for account2: "; // prompt
31      std::cin >> amount; // obtain user input
32      std::cout << fmt::format(
33         "adding ${:.2f} to account2 balance\n\n", amount);
34      account2.deposit(amount); // add to account2 balance
35
36      // display balances
37      std::cout << fmt::format("account1: {} balance is ${:.2f}\n",
38         account1.getName(), account1.getBalance());
39      std::cout << fmt::format("account2: {} balance is ${:.2f}\n",
40         account2.getName(), account2.getBalance());
41   }
```

```
Enter deposit amount for account2: 123.45
adding $123.45 to account2 balance

account1: Jane Green balance is $75.37
account2: John Blue balance is $123.45
```

9.7 `Time` Class Case Study: Separating Interface from Implementation

Each of our prior custom class definitions placed a class in a header for reuse, then included the header into a source-code file containing `main`, so we could create and manipulate objects of the class. Unfortunately, placing a complete class definition in a header reveals the class's entire implementation to its clients. A header is simply a text file that anyone can open and read.

SE Conventional software-engineering wisdom says that to use an object of a class, the client code (e.g., `main`) needs to know only

- which member functions to call,
- which arguments to provide to each member function and
- which return type to expect from each member function.

The client code does not need to know how those functions are implemented. This is another example of the principle of least privilege.

SE If the client-code programmer knows how a class is implemented, the programmer might write client code based on the class's implementation details. **If that implementation changes, the class's clients should not have to change.** Hiding the class's implementation details makes it easier to change the implementation while minimizing and hopefully eliminating changes to client code.

Our next example creates and manipulates an object of class `Time`.[14] We demonstrate two important C++ software engineering concepts:

14. Rather than building your own classes to represent times and dates, you'll typically use capabilities from the C++ standard library header `<chrono>` (https://en.cppreference.com/w/cpp/chrono).

- Separating interface from implementation.

- Using the preprocessor directive "`#pragma once`" in a header to prevent the header code from being included in the same source-code file more than once. Since a class can be defined only once, using "`#pragma once`" prevents multiple-definition errors.

20 C++20 Modules Change How You Separate Interface from Implementation

Mod As you'll see in Chapter 16, C++20 modules[15] eliminate the need for preprocessor directives like `#pragma once`. You'll also see that modules enable you to separate interface from implementation in a single source-code file or by using multiple source-code files.

15. At the time of this writing, the C++20 modules features were not fully implemented by the three compilers we use, so we cover them in a separate chapter.

9.7.1 Interface of a Class

Interfaces define and standardize how people and systems interact with one another. For example, a radio's controls serve as an interface between users and its internal components. The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differently—some provide buttons, some provide dials and some support voice commands. The interface specifies *what* operations a radio permits users to perform but does not specify *how* the operations are implemented inside the radio.

Similarly, the **interface of a class** describes *what* services a class's clients can use and how to request those services, but not *how* the class implements them. A class's `public` interface consists of the class's `public` member functions (also known as the class's `public` **services**). As you'll soon see, you can specify a class's interface by writing a class definition that lists only the class's member-function prototypes in the class's `public` section.

9.7.2 Separating the Interface from the Implementation

To separate the class's interface from its implementation, we break up class `Time` into two files—`Time.h` (Fig. 9.7) defines class `Time` and `Time.cpp` (Fig. 9.8) defines class `Time`'s member functions. This split

- helps make the class reusable,
- ensures that the clients of the class know what member functions the class provides, how to call them and what return types to expect, and
- enables the clients to ignore how the class's member functions are implemented.

Perf In addition, this split can reduce compilation time because the implementation file can be compiled, then does not need to be recompiled unless the implementation changes.

By convention, member-function definitions are placed in a `.cpp` file with the same base name (e.g., `Time`) as the class's header. Some compilers support other filename extensions as well. Figure 9.9 defines function `main`, which uses objects of our `Time` class.

9.7.3 Class Definition

The header `Time.h` (Fig. 9.7) contains `Time`'s class definition (lines 8–17). Rather than function definitions, the class contains function prototypes (lines 10–12) that describe the class's `public` interface without revealing the member-function implementations. The function prototype in line 10 indicates that `setTime` requires three `int` parameters and returns `void`. The prototypes for `to24HourString` and `to12HourString` (lines 11–12) specify that they take no arguments and return a `string`. Classes with one or more constructors would also declare them in the header, as we'll do in subsequent examples.

```
1  // Fig. 9.7: Time.h
2  // Time class definition.
3  // Member functions are defined in Time.cpp
4  #pragma once // prevent multiple inclusions of header
5  #include <string>
6
7  // Time class definition
8  class Time {
9  public:
10    void setTime(int hour, int minute, int second);
11    std::string to24HourString() const; // 24-hour string format
12    std::string to12HourString() const; // 12-hour string format
13 private:
14    int m_hour{0}; // 0 - 23 (24-hour clock format)
15    int m_minute{0}; // 0 - 59
16    int m_second{0}; // 0 - 59
17 };
```

Fig. 9.7 `Time` class definition.

11 The header still specifies the class's `private` data members (lines 14–16). Each uses a C++11 **in-class initializer** to set the data member to `0`. The compiler must know the class's data members to determine how much memory to reserve for each object of the class. Including the header `Time.h` in the client code provides the compiler with the information it needs to ensure that the client code calls class `Time`'s member functions correctly.

CG CG Perf The C++ Core Guidelines recommend using in-class initializers when a data member should be initialized with a constant.[16] The Core Guidelines also recommend that, when possible, you initialize all your data members with in-class initializers and let the compiler generate a default constructor for your class. A compiler-generated default constructor can be more efficient than one you define.[17]

16. C++ Core Guidelines, "C.48: Prefer In-Class Initializers to Member Initializers in Constructors for Constant Initializers." Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-in-class-initializer.
17. C++ Core Guidelines, "C.45: Don't Define a Default Constructor That Only Initializes Data Members; Use In-Class Member Initializers Instead." Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-default.

`#pragma once`

Err Mod In larger programs, headers also will contain other definitions and declarations. Attempts to include a header multiple times (inadvertently) often occur in programs with many headers that might include other headers. This could lead to compilation errors if the same definition appears more than once in a preprocessed file. The `#pragma once` directive (line 4) prevents `time.h`'s contents from being included in the same source-code file more than once. This is sometimes referred to as an **include guard**. In Chapter 16, we'll discuss how C++20 modules help prevent such problems.

9.7.4 Member Functions

`Time.cpp` (Fig. 9.8) defines class `Time`'s member functions, which were declared in lines 10–12 of Fig. 9.7. For member functions `to24HourString` and `to12HourString`, the `const` keyword must appear in both the function prototypes (Fig. 9.7, lines 11–12) and the function definitions (Fig. 9.8, lines 23 and 29).

```
1  // Fig. 9.8: Time.cpp
2  // Time class member-function definitions.
3  #include <fmt/format.h>
4  #include <stdexcept> // for invalid_argument exception class
5  #include <string>
6  #include "Time.h" // include definition of class Time from Time.h
7
8  // set new Time value using 24-hour time
9  void Time::setTime(int hour, int minute, int second) {
10    // validate hour, minute and second
11    if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
12       (second < 0 || second >= 60)) {
13       throw std::invalid_argument{
14          "hour, minute or second was out of range"};
15    }
16
17    m_hour = hour;
18    m_minute = minute;
19    m_second = second;
20 }
21
22 // return Time as a string in 24-hour format (HH:MM:SS)
23 std::string Time::to24HourString() const{
24    return fmt::format("{:02d}:{:02d}:{:02d}", m_hour, m_minute, m_second);
25 }
26
27 // return Time as string in 12-hour format (HH:MM:SS AM or PM)
28 std::string Time::to12HourString() const {
29    return fmt::format("{}:{:02d}:{:02d} {}",
30             ((m_hour % 12 == 0) ? 12 : m_hour % 12), m_minute, m_second,
31              (m_hour < 12 ? "AM" : "PM"));
32 }
```

Fig. 9.8 `Time` class member-function definitions.

9.7.5 Including the Class Header in the Source-Code File

To indicate that the member functions in `Time.cpp` are part of class `Time`, we must first include the `Time.h` header (Fig. 9.8, line 6). This allows us to use the class name `Time` in the `Time.cpp` file (lines 9, 23 and 28). When compiling `Time.cpp`, the compiler uses the information in `Time.h` to ensure that

- the first line of each member function matches its prototype in `Time.h` and
- each member function knows about the class's data members and other member functions.

9.7.6 Scope Resolution Operator (`::`)

Each member function's name (lines 9, 23 and 28) is preceded by the class name and the scope resolution operator (`::`). This "ties" them to the (now separate) `Time` class definition (Fig. 9.7), which declares the class's members. The `Time::` tells the compiler that each member function is in that **class's scope**, and its name is known to other class members.

Without "`Time::`" preceding each function name, the compiler would treat these as global functions with no relationship to class `Time`. Such functions, also called "**free**" functions, cannot access `Time`'s `private` data or call the class's member functions without specifying an object. So, the compiler would not be able to compile these functions because it would not know that class `Time` declares variables `m_hour`, `m_minute` and `m_second`. In Fig. 9.8, lines 17–19, 24 and 30–31 would cause compilation errors because `m_hour`, `m_minute` and `m_second` are not declared as local variables in each function, nor are they declared as global variables.

9.7.7 Member Function `setTime` and Throwing Exceptions

Function `setTime` (lines 9–20) is a `public` function that declares three `int` parameters and uses them to set the time. Lines 11–12 test each argument to determine whether the value is in range. If so, lines 17–19 assign the values to the `m_hour`, `m_minute` and `m_second` data members, respectively. The `hour` argument must be greater than or equal to `0` and less than `24` because the 24-hour time format represents hours as integers from 0 to 23. Similarly, the `minute` and `second` arguments must be greater than or equal to `0` and less than `60`.

If any of the values is outside its range, `setTime` **throws an exception** (lines 13–14) of type `invalid_argument` (header `<stdexcept>`), notifying the client code that an invalid argument was received. As you saw in Section 6.15, you can use `try` ... `catch` to catch exceptions and attempt to recover from them, which we'll do in Fig. 9.9. The `throw` **statement** creates a new `invalid_argument` object, initializing it with a custom error message string. After the exception object is created, the `throw` statement terminates function `set-Time`. Then, the exception is returned to the code that called `setTime`.

Invalid values cannot be stored in a `Time` object because

- when a `Time` object is created, its default constructor is called, and each data member is initialized to 0, as specified in lines 14–16 of Fig. 9.7—this is the equivalent of 12 AM (midnight)—and
- all subsequent attempts by a client to modify the data members are scrutinized by function `setTime`.

9.7.8 Member Functions `to24HourString` and `to12HourString`

Member function `to24HourString` (lines 23–25 of Fig. 9.8) takes no arguments and returns a formatted 24-hour `string` with three colon-separated digit pairs. So, if the time is 1:30:07 PM, the function returns `"13:30:07"`. Each `{:02d}` placeholder in line 24 formats an integer (`d`) in a field width of two.

The `0` before the field width indicates that values with fewer than two digits should be formatted with leading zeros.

Function `to12HourString` (lines 28–32) takes no arguments and returns a formatted 12-hour time `string` containing the `m_hour`, `m_minute` and `m_second` values separated by colons and followed by an AM or PM indicator (e.g., `10:54:27 AM` and `1:27:06 PM`). The function uses the placeholder `{:02d}` to format `m_minute` and `m_second` as two-digit values with leading zeros, if necessary. Line 30 uses the conditional operator (`?:`) to determine how `m_hour` should be formatted. If `m_hour` is `0` or `12` (AM or PM, respectively), it appears as 12; otherwise, we use the remainder operator (`%`) to get a value from 1 to 11. The conditional operator in line 31 determines whether to include AM or PM.

9.7.9 Implicitly Inlining Member Functions

Perf If a member function is fully defined in a class's body (as we did in our `Account` class examples), the member function is implicitly declared `inline` (Section 5.11). This can improve performance. Remember that the compiler reserves the right not to inline any function. Similarly, optimizing compilers also reserve the right to inline functions even if they are not declared with the `inline` keyword, provided that the compiler has access to the function's definition.

Perf Only the simplest, most stable member functions (i.e., whose implementations are unlikely to change) and those that are the most performance-sensitive should be defined in the class header. Every change to the header requires you to recompile every source-code file dependent on that header—a time-consuming task in large systems.

9.7.10 Member Functions vs. Global Functions

SE Member functions `to24HourString` and `to12HourString` take no arguments. They implicitly know about and can access the data members for the `Time` object on which they're invoked. This is a nice benefit of object-oriented programming. In general, member-function calls receive either no arguments or fewer arguments than function calls in non-object-oriented programs. This reduces the likelihood of passing wrong arguments, the wrong number of arguments or arguments in the wrong order.

9.7.11 Using Class `Time`

Once class `Time` is defined, it can be used as a type in declarations, as in:

```
Time sunset{}; // object of type Time
std::array<Time, 5> arrayOfTimes{}; // std::array of 5 Time objects
Time& dinnerTimeRef{sunset}; // reference to a Time object
Time* timePtr{&sunset}; // pointer to a Time object
```

Figure 9.9 creates and manipulates a `Time` object. Separating `Time`'s interface from the implementation of its member functions does not affect how this client code uses the class. Line 8 includes `Time.h` so the compiler knows how much space to reserve for the `Time` object `t` (line 17) and can ensure that `Time` objects are created and manipulated correctly in the client code.

```
1  // fig09_09.cpp
2  // Program to test class Time.
```

```
3  // NOTE: This file must be linked with Time.cpp.
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <stdexcept> // invalid_argument exception class
7  #include <string_view>
8  #include "Time.h" // definition of class Time from Time.h
9
10 // displays a Time in 24-hour and 12-hour formats
11 void displayTime(std::string_view message, const Time& time) {
12    std::cout << fmt::format("{}\n24-hour time: {}\n12-hour time: {}\n\n",
13       message, time.to24HourString(), time.to12HourString());
14 }
15
16 int main() {
17    Time t{}; // instantiate object t of class Time
18
19    displayTime("Initial time:", t); // display t's initial value
20    t.setTime(13, 27, 6); // change time
21    displayTime("After setTime:", t); // display t's new value
22
23    // attempt to set the time with invalid values
24    try {
25       t.setTime(99, 99, 99); // all values out of range
26    }
27    catch (const std::invalid_argument& e) {
28       std::cout << fmt::format("Exception: {}\n\n", e.what());
29    }
30
31    // display t's value after attempting to set an invalid time
32    displayTime("After attempting to set an invalid time:", t);
33 }
```

```
Initial time:
24-hour time: 00:00:00
12-hour time: 12:00:00 AM

After setTime:
24-hour time: 13:27:06
12-hour time: 1:27:06 PM

Exception: hour, minute and/or second was out of range

After attempting to set an invalid time:
24-hour time: 13:27:06
12-hour time: 1:27:06 PM
```

Fig. 9.9 Program to test class `Time`.

Throughout the program, we display `string` representations of the `Time` object using function `displayTime` (lines 11–14), which calls `Time` member functions `to24HourString` and `to12HourString`. Line 17 creates the `Time` object `t`. Recall that class `Time` does not define a constructor, so this statement calls the compiler-generated default constructor. Thus, `t`'s `m_hour`, `m_minute` and `m_second` are set to `0` via their initializers in class `Time`'s definition. Then, line 19 displays the time in 24-hour and 12-hour formats, respectively, to confirm that the members were correctly initialized. Line 20 sets a new valid time by calling member function `setTime`, and line 21 again shows the time in both formats.

Calling `setTime` with Invalid Values

To show that `setTime` validates its arguments, line 25 calls `setTime` with invalid arguments of `99` for the `hour`, `minute` and `second` parameters. We placed this statement in a `try` block (lines 24–26) in case `setTime` throws an `invalid_argument` exception, which it will do in this example. When the exception occurs, it's caught at lines 27–29, and line 28 displays the exception's error message by calling its `what` member function. Line 32 shows the time to confirm that `setTime` did not change the time when invalid arguments were supplied.

9.7.12 Object Size

People new to object-oriented programming often suppose that objects must be quite large because they contain data members and member functions. Logically, this is true. You may think of objects as containing data and functions physically (and our discussion has certainly encouraged this view). However, this is not the case.

An object in memory contains only data, not the class's member functions. The member functions' code is maintained separately from all objects of the class. Each object needs its own data because the data usually varies among the objects. The function code is the same for all objects of the class and can be shared among them.

9.8 Compilation and Linking Process

Often a class's interface and implementation will be created by one programmer and used by a separate programmer who implements the client code. A **class-implementation programmer** responsible for creating a reusable `Time` class creates the header `Time.h` and the source-code file `Time.cpp` that `#include`s the header, then provides these files to the client-code programmer. A reusable class's source code often is available to client-code programmers as a library they can download from a website, like `github.com`.

The client-code programmer needs to know only `Time`'s interface to use the class and must be able to compile `Time.cpp` and link its object code. Since the class's interface is part of the class definition in the `Time.h` header, the client-code programmer must `#include` this file in the client's source-code file. The compiler uses the class definition in `Time.h` to ensure that the client code correctly creates and manipulates `Time` objects.

To create the executable `Time` application, the last step is to link

- the object code for the `main` function (that is, the client code),
- the object code for class `Time`'s member-function implementations and
- the C++ standard library object code for the C++ classes (such as `std::string`) used by the class-implementation programmer and the client-code programmer.

SE The linker's output for the program of Section 9.7 is the executable application that users can run to create and manipulate a `Time` object. Compilers and IDEs typically invoke the linker for you after compiling your code.

Compiling Programs Containing Two or More Source-Code Files

Section 1.2 showed how to compile and run C++ applications that contained one source-code (`.cpp`) file. To compile and link multiple source-code files:[18]

Mod 18. This process changes with C++20 modules, as we'll discuss in Chapter 16.

- In Microsoft Visual Studio, add to your project (as shown in Section 1.2.1) all the custom headers and source-code files that make up the program, then build and run the project. You can place the headers in the project's `Header Files` folder and the source-code files in the project's `Source Files` folder, but these are mainly for organizing files in large projects. The programs will compile if you place all the files in the `Source Files` folder.
- For `g++` or `clang++` at the command line, open a shell and change to the directory containing all the files for a given program. Then in your compilation command, either list each `.cpp` file by name or use `*.cpp` to compile all the `.cpp` files in the current folder. The preprocessor automatically locates the program-specific headers in that folder.
- For Apple Xcode, add to your project (as shown in Section 1.2.2) all the headers and source-code files that make up a program, then build and run the project.

9.9 Class Scope and Accessing Class Members

A class's data members and member functions belong to that class's scope. Non-member functions are defined at global namespace scope by default. (We discuss namespaces in more detail in online Chapter 20.) Within a class's scope, class members are immediately accessible by all of that class's member functions and can be referenced by name. Outside a class's scope, `public` class members are referenced through

- an object name,
- a reference to an object,
- a pointer to an object or
- a pointer to a specific class member (discussed briefly in online Chapter 20).

We refer to these as **handles** on an object. The handle's type helps the compiler determine the interface (that is, the member functions) accessible to the client via that handle. We'll see in Section 9.19 that an implicit handle (called the `this` pointer) is inserted by the compiler each time you refer to a data member or member function from within an object.

Dot (`.`) and Arrow (`->`) Member-Selection Operators

As you know, you can use an object's name or a reference to an object followed by the dot member-selection operator (`.`) to access the object's members. To reference an object's members via a pointer to an object, follow the pointer name by the **arrow member-selection operator (`->`)** and the member name, as in *pointerName*`->`*memberName*.

Accessing `public` Class Members Through Objects, References and Pointers

Consider an `Account` class that has a `public deposit` member function. Given the following declarations:

```
Account account{}; // an Account object
Account& ref{account}; // ref refers to an Account object
Account* ptr{&account}; // ptr points to an Account object
```

you can invoke member function `deposit` using the dot (`.`) and arrow (`->`) member selection operators as follows:

```
account.deposit(123.45); // call deposit via account object's name
ref.deposit(123.45); // call deposit via reference to account object
ptr->deposit(123.45); // call deposit via pointer to account object
```

Again, you should use references in preference to pointers whenever possible. We'll continue to show pointers when they are required and to prepare you to work with them in the legacy code you'll encounter in industry.

9.10 Access Functions and Utility Functions

Access Functions

Access functions can read or display data, but not modify it. Another use of access functions is to test whether a condition is true or false. Such functions are often called **predicate functions.** An example would be a `std::array`'s or a `std::vector`'s `empty` function. A program might test `empty` before attempting to read an item from the container object.[19]

19. Many programmers prefer to begin the names of predicate functions with the word "`is`." For example, useful predicate functions for our `Time` class might be `isAM` and `isPM`. Such functions also should be preceded with the attribute `[[nodiscard]]` so the compiler confirms that the return value is used in the caller, rather than ignored.

Utility Functions

A **utility function** (also called a **helper function**) is a `private` member function that supports the operation of a class's other member functions and is not intended for use by the class's clients. Typically, a utility function contains code that would otherwise be duplicated in several other member functions. Many codebases have naming conventions for `private` member functions, such as preceding their names with an underscore (_).

9.11 Time Class Case Study: Constructors with Default Arguments

The program of Figs. 9.10–9.12 enhances class `Time` to demonstrate a constructor with default arguments.

9.11.1 Class `Time`

Like other functions, constructors can specify default arguments. Line 11 of Fig. 9.10 declares a `Time` constructor with the default argument value `0` for each parameter. A constructor with default arguments for all its parameters is also a default constructor—that is, it can be invoked with no arguments.[20] There can be at most one default constructor per class. Any change to a function's default argument values requires the client code to be recompiled (to ensure that the program still functions correctly).

20. You'll see that C++20 concepts (Chapter 15) enable you to overload any function or member function for use with types that match specific requirements.

```
1  // Fig. 9.10: Time.h
2  // Time class containing a constructor with default arguments.
3  // Member functions defined in Time.cpp.
4  #pragma once // prevent multiple inclusions of header
5  #include <string>
6
```

```
// Time class definition
class Time {
public:
   // default constructor because it can be called with no arguments
   explicit Time(int hour = 0, int minute = 0, int second = 0);

   // set functions
   void setTime(int hour, int minute, int second);
   void setHour(int hour); // set hour (after validation)
   void setMinute(int minute); // set minute (after validation)
   void setSecond(int second); // set second (after validation)

   // get functions
   int getHour() const; // return hour
   int getMinute() const; // return minute
   int getSecond() const; // return second

   std::string to24HourString() const; // 24-hour time format string
   std::string to12HourString() const; // 12-hour time format string
privat:
   int m_hour{0}; // 0 - 23 (24-hour clock format)
   int m_minute{0}; // 0 - 59
   int m_second{0}; // 0 - 59
};
```

Fig. 9.10 `Time` class containing a constructor with default arguments.

Class `Time`'s Constructor

Err In Fig. 9.11, lines 9–11 define the `Time` constructor, which calls `setTime` to validate and assign values to the data members. Function `setTime` (lines 14–31) ensures that `hour` is in the range 0–23 and `minute` and `second` are each in the range 0–59. If any argument is out of range, `setTime` throws an exception—in which case, the `Time` object will not complete construction and will not exist for use in the program. This version of `setTime` uses separate `if` statements to validate the arguments so we can provide precise error messages, indicating which argument is out of range.
Functions `setHour`, `setMinute` and `setSecond` (lines 34–40) each call `setTime`. Each passes its argument and the current values of the other two data members. For example, `setHour` passes its `hour` argument to change `m_hour`, and the current values of `m_minute` and `m_second`.

```
// Fig. 9.11: Time.cpp
// Member-function definitions for class Time.
#include <fmt/format.h>
#include <stdexcept>
#include <string>
#include "Time.h" // include definition of class Time from Time.h

// Time constructor initializes each data member
Time::Time(int hour, int minute, int second) {
   setTime(hour, minute, second);
}

// set new Time value using 24-hour time
void Time::setTime(int hour, int minute, int second) {
```

```
15     // validate hour, minute and second
16     if (hour < 0 || hour >= 24) {
17        throw std::invalid_argument{"hour was out of range"};
18     }
19
20     if (minute < 0 || minute >= 60) {
21        throw std::invalid_argument{"minute was out of range"};
22     }
23
24     if (second < 0 || second >= 60) {
25        throw std::invalid_argument{"second was out of range"};
26     }
27
28     m_hour = hour;
29     m_minute = minute;
30     m_second = second;
31  }
32
33  // set hour value
34  void Time::setHour(int hour) {setTime(hour, m_minute, m_second);}
35
36  // set minute value
37  void Time::setMinute(int minute) {setTime(m_hour, minute, m_second);}
38
39  // set second value
40  void Time::setSecond(int second) {setTime(m_hour, m_minute, second);}
41
42  // return hour value
43  int Time::getHour() const {return m_hour;}
44
45  // return minute value
46  int Time::getMinute() const {return m_minute;}
47
48  // return second value
49  int Time::getSecond() const {return m_second;}
50
51  // return Time as a string in 24-hour format (HH:MM:SS)
52  std::string Time::to24HourString() const{
53     return fmt::format("{:02d}:{:02d}:{:02d}",
54        getHour(), getMinute(), getSecond());
55  }
56
57  // return Time as a string in 12-hour format (HH:MM:SS AM or PM)
58  std::string Time::to12HourString() const {
59     return fmt::format("{}:{:02d}:{:02d} {}",
60        ((getHour() % 12 == 0) ? 12 : getHour() % 12),
61        getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
62  }
```

Fig. 9.11 Member-function definitions for class `Time`.

CG Err The C++ Core Guidelines provide many constructor recommendations. We'll see more in the next two chapters. If a class has a **class invariant**[21]—that is, it requires its data members to have specific values or ranges of values (as in class `Time`)—the class should define a constructor that validates its arguments and, if any are invalid, throws an exception to prevent the object from being created.[22,23,24]

21. “Class invariant.” Wikipedia. Wikimedia Foundation. Accessed January 8, 2022. https://en.wikipedia.org/wiki/Class_invariant.
22. C++ Core Guidelines, “C.40: Define a Constructor if a Class Has an Invariant.” Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor.
23. C++ Core Guidelines, “C.41: A Constructor Should Create a Fully Initialized Object.” Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-complete.
24. C++ Core Guidelines, “C.42: If a Constructor Cannot Construct a Valid Object, Throw an Exception.” Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-throw.

Testing the Updated Class Time

Function `main` in Fig. 9.12 initializes five `Time` objects:

- one with all three arguments defaulted in the implicit constructor call (line 16),
- one with one argument specified (line 17),
- one with two arguments specified (line 18),
- one with three arguments specified (line 19) and
- one with three invalid arguments specified (line 29).

```
1  // fig09_12.cpp
2  // Constructor with default arguments.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <stdexcept>
6  #include <string>
7  #include "Time.h" // include definition of class Time from Time.h
8
9  // displays a Time in 24-hour and 12-hour formats
10 void displayTime(std::string_view message, const Time& time) {
11    std::cout << fmt::format("{}\n24-hour time: {}\n12-hour time: {}\n\n",
12       message, time.to24HourString(), time.to12HourString());
13 }
14
15 int main() {
16    const Time t1{}; // all arguments defaulted
17    const Time t2{2}; // hour specified; minute & second defaulted
18    const Time t3{21, 34}; // hour & minute specified; second defaulted
19    const Time t4{12, 25, 42}; // hour, minute & second specified
20
21    std::cout << "Constructed with:\n\n";
22    displayTime("t1: all arguments defaulted", t1);
23    displayTime("t2: hour specified; minute and second defaulted", t2);
24    displayTime("t3: hour and minute specified; second defaulted", t3);
25    displayTime("t4: hour, minute and second specified", t4);
26
27    // attempt to initialize t5 with invalid values
28    try {
29       const Time t5{27, 74, 99}; // all bad values specified
30    }
31    catch (const std::invalid_argument& e) {
32       std::cerr << fmt::format("t5 not created: {}\n", e.what());
33    }
34 }
```

```
Constructed with:

t1: all arguments defaulted
24-hour time: 00:00:00
12-hour time: 12:00:00 AM

t2: hour specified; minute and second defaulted
24-hour time: 02:00:00
12-hour time: 2:00:00 AM

t3: hour and minute specified; second defaulted
24-hour time: 21:34:00
12-hour time: 9:34:00 PM

t4: hour, minute and second specified
24-hour time: 12:25:42
12-hour time: 12:25:42 PM

t5 not created: hour was out of range
```

Fig. 9.12 Constructor with default arguments.

Err The program displays each object in 24-hour and 12-hour time formats. For `Time` object `t5` (line 29), the program displays an error message because the constructor arguments are out of range. The variable `t5` never represents a fully constructed object in this program because the exception is thrown during construction.

SE Software Engineering Notes Regarding Class `Time`'s *set* and *get* Functions and Constructor

`Time`'s *set* and *get* functions are called throughout the class's body. In particular, the constructor (Fig. 9.11, lines 9–11) calls `setTime`, and `to24HourString` and `to12HourString` call `getHour`, `getMinute` and `getSecond` in lines 54 and lines 60–61. In each case, we could have accessed the class's `private` data directly.

SE Our internal time representation uses three `int`s, requiring 12 bytes of memory on systems with four-byte `int`s. Consider changing this to the total seconds since midnight—a single `int` requiring only four bytes of memory. If we made this change, only the bodies of functions that directly access the `private` data would need to change. In this class, we'd modify `setTime` and the *set* and *get* function's bodies for `m_hour`, `m_minute` and `m_second`. There would be no need to modify the constructor or functions `to24HourString` or `to12HourString` because they do not access the data directly.

Err Duplicating statements in multiple functions or constructors makes changing the class's internal data representation more difficult. Implementing the `Time` constructor and functions `to24HourString` and `to12HourString`, as shown in this example, reduces the likelihood of errors when altering the class's implementation.

SE **As a general rule: Avoid repeating code. This principle is referred to as DRY—"don't repeat yourself."**[25] Rather than duplicating code, place it in a member function that can be called by the class's constructor or other member functions. This simplifies code maintenance and reduces the likelihood of an error if the code implementation is modified.

25. “Don't Repeat Yourself.” Wikipedia. Wikimedia Foundation. Accessed January 6, 2022, https://en.wikipedia.org/wiki/Don't_repeat_yourself.

SE Err A constructor can call the class’s other member functions. You must be careful when doing this. The constructor initializes the object, so data members used in the called function may not yet be initialized. Logic errors may occur if you use data members before they have been properly initialized.

SE Making data members `private` and controlling access (especially write access) to those data members through `public` member functions helps ensure data integrity. The benefits of data integrity are not automatic simply because data members are `private`. You must provide appropriate validity checking.

11 9.11.2 Overloaded Constructors and C++11 Delegating Constructors

Section 5.15 showed how to overload functions. A class’s constructors and member functions also can be overloaded. Overloaded constructors allow objects to be initialized with different types and/or numbers of arguments. To overload a constructor, provide a prototype and definition for each overloaded version. This also applies to overloaded member functions.

In Figs. 9.10–9.12, class `Time`’s constructor had a default argument for each parameter. We could have defined that constructor instead as four overloaded constructors with the following prototypes:

```
Time(); // default m_hour, m_minute and m_second to 0
explicit Time(int hour); // default m_minute & m_second to 0
Time(int hour, int minute); // default m_second to 0
Time(int hour, int minute, int second); // no default values
```

11 CG Just as a constructor can call a class’s other member functions to perform tasks, constructors can call other constructors in the same class. The calling constructor is known as a **delegating constructor** (C++11)—it delegates its work to another constructor. The C++ Core Guidelines recommend defining common code for overloaded constructors in one constructor, then using delegating constructors to call it.[26] Before C++11, this would have been accomplished via a `private` utility function called by all the constructors.

26. C++ Core Guidelines, “C.51: Use Delegating Constructors to Represent Common Actions for all Constructors of a Class.” Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-delegating.

The first three of the four `Time` preceding constructor declarations can delegate work to one with three `int` arguments, passing `0` as the default value for the extra parameters. To do so, you use a member initializer with the name of the class, as in:

```
Time::Time() : Time{0, 0, 0} {}

Time::Time(int hour) : Time{hour, 0, 0} {}

Time::Time(int hour, int minute) : Time{hour, minute, 0} {}
```

9.12 Destructors

A **destructor** is a special member function that may not specify parameters or a return type. A class's destructor name is the **tilde character (~)** followed by the class name, such as `~Time`. This naming convention has intuitive appeal because, as we'll see in a later chapter, the tilde is the bitwise complement operator. In a sense, the destructor is the complement of the constructor.

A class's destructor is called implicitly when an object is destroyed, typically when program control leaves the scope in which that object was created. The destructor itself does not actually remove the object from memory. It performs **termination housekeeping** (such as closing a file and cleaning up other resources used by the object) before the object's memory is reclaimed for later use.

Even though destructors have not been defined for the classes presented so far, every class has exactly one destructor. If you do not explicitly define a destructor, the compiler defines a default destructor that invokes any class-type data members' destructors.[27] In Chapter 12, we'll explain why exceptions should not be thrown from destructors.

27. We'll see that such a default destructor also destroys class objects that are created through inheritance (Chapter 10).

9.13 When Constructors and Destructors Are Called

SE Constructors and destructors are called implicitly when objects are created and when they're about to go out of scope, respectively. The order in which these are called depends on the objects' scopes. Generally, destructor calls are made in the reverse order of the corresponding constructor calls, but as we'll see in Figs. 9.13–9.15, global and `static` objects can alter the order in which destructors are called.

Constructors and Destructors for Objects in Global Scope

SE Err Err Constructors are called for objects defined in the global scope (also called global namespace scope) before executing any other function in that file. The execution order of global object constructors among multiple files is not guaranteed, so global objects in separate files should not depend on one another. When `main` terminates via a `return` statement or by reaching its closing brace, the corresponding destructors are called in the reverse order of their construction. The `exit` function often is used to terminate a program when a fatal unrecoverable error occurs. Function **`exit`** forces a program to terminate immediately and does not execute the destructors of local objects.[28] Function **`abort`** performs similarly to function `exit` but forces the program to terminate immediately, without allowing programmer-defined cleanup code of any kind to be called. Function `abort` is usually used to indicate abnormal program termination.[29]

28. "`std::exit`." Accessed January 6, 2022. https://en.cppreference.com/w/cpp/utility/program/exit.

29. "`std::abort`." Accessed January 6, 2022. https://en.cppreference.com/w/cpp/utility/program/abort.

Constructors and Destructors for Non-`static` Local Objects

A non-`static` local object's constructor is called when execution reaches the object's definition. Its destructor is called when execution leaves the object's scope—that is, when the block in which that object is defined finishes executing normally or due to an exception. Destructors are not called for local objects if the program terminates with a call to function `exit` or `abort`.

Constructors and Destructors for `static` Local Objects

The constructor for a `static` local object is called only once when execution first reaches the point where the object is defined. The corresponding destructor is called when `main` terminates, or the program calls function `exit`. Global and `static` objects are destroyed in the reverse order of their creation. Destructors are not called for `static` objects if the program terminates with a call to function `abort`.

Demonstrating When Constructors and Destructors Are Called

The program in Figs. 9.13–9.15 demonstrates the order in which constructors and destructors are called for global, local and local `static` objects of class `CreateAndDestroy` (Fig. 9.13 and Fig. 9.14). This mechanical example is purely for pedagogic purposes.

Figure 9.13 declares class `CreateAndDestroy`. Lines 13–14 declare the class's data members—an integer (`m_ID`) and a `string` (`m_message`) to identify each object in the program's output.

```
1  // Fig. 9.13: CreateAndDestroy.h
2  // CreateAndDestroy class definition.
3  // Member functions defined in CreateAndDestroy.cpp.
4  #pragma once // prevent multiple inclusions of header
5  #include <string>
6  #include <string_view>
7
8  class CreateAndDestroy {
9  public:
10    CreateAndDestroy(int ID, std::string_view message); // constructor
11    ~CreateAndDestroy(); // destructor
12 private:
13    int m_ID; // ID number for object
14    std::string m_message; // message describing object
15 };
```

Fig. 9.13 `CreateAndDestroy` class definition.

The constructor and destructor implementations (Fig. 9.14) both display lines of output to indicate when they're called. In the destructor, the conditional expression (line 18) determines whether the object being destroyed has the `m_ID` value `1` or `6` and, if so, outputs a newline character to make the program's output easier to follow.

```
1  // Fig. 9.14: CreateAndDestroy.cpp
2  // CreateAndDestroy class member-function definitions.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include "CreateAndDestroy.h"// include CreateAndDestroy class definition
6
7  // constructor sets object's ID number and descriptive message
8  CreateAndDestroy::CreateAndDestroy(int ID, std::string_view message)
9     : m_ID{ID}, m_message{message} {
10    std::cout << fmt::format("Object {} constructor runs {}\n",
11       m_ID, m_message);
12 }
13
14 // destructor
15 CreateAndDestroy::~CreateAndDestroy() {
```

```
16     // output newline for certain objects; helps readability
17     std::cout << fmt::format("{}Object {} destructor runs {}\n",
18        (m_ID == 1 || m_ID == 6 ? "\n" : ""), m_ID, m_message);
19  }
```

Fig. 9.14 `CreateAndDestroy` class member-function definitions.

Figure 9.15 defines the object `first` (line 9) in the global scope. Its constructor is called before any statements in `main` execute, and its destructor is called at program termination after the destructors for all objects with automatic storage duration have run.

```
 1  // fig09_15.cpp
 2  // Order in which constructors and
 3  // destructors are called.
 4  #include <iostream>
 5  #include "CreateAndDestroy.h" // include CreateAndDestroy class definition
 6
 7  void create(); // prototype
 8
 9  const CreateAndDestroy first{1, "(global before main)"}; // global object
10
11  int main() {
12     std::cout << "\nMAIN FUNCTION: EXECUTION BEGINS\n";
13     const CreateAndDestroy second{2, "(local in main)"};
14     static const CreateAndDestroy third{3, "(local static in main)"};
15
16     create(); // call function to create objects
17
18     std::cout << "\nMAIN FUNCTION: EXECUTION RESUMES\n";
19     const CreateAndDestroy fourth{4, "(local in main)"};
20     std::cout << "\nMAIN FUNCTION: EXECUTION ENDS\n";
21  }
22
23  // function to create objects
24  void create() {
25     std::cout << "\nCREATE FUNCTION: EXECUTION BEGINS\n";
26     const CreateAndDestroy fifth{5, "(local in create)"};
27     static const CreateAndDestroy sixth{6, "(local static in create)"};
28     const CreateAndDestroy seventh{7, "(local in create)"};
29     std::cout << "\nCREATE FUNCTION: EXECUTION ENDS\n";
30  }
```

```
Object 1   constructor runs   (global before main)

MAIN FUNCTION: EXECUTION BEGINS
Object 2   constructor runs   (local in main)
Object 3   constructor runs   (local static in main)

CREATE FUNCTION: EXECUTION BEGINS
Object 5   constructor runs   (local in create)
Object 6   constructor runs   (local static in create)
Object 7   constructor runs   (local in create)

CREATE FUNCTION: EXECUTION ENDS
Object 7   destructor runs    (local in create)
Object 5   destructor runs    (local in create)
```

```
MAIN FUNCTION: EXECUTION RESUMES
Object 4   constructor runs   (local in main)

MAIN FUNCTION: EXECUTION ENDS
Object 4   destructor runs   (local in main)
Object 2   destructor runs   (local in main)

Object 6   destructor runs   (local static in create)
Object 3   destructor runs   (local static in main)

Object 1   destructor runs (global before main)
```

Fig. 9.15 Order in which constructors and destructors are called.

Function `main` (lines 11–21) defines three objects. Objects `second` (line 13) and `fourth` (line 19) are local objects, and object `third` (line 14) is a `static` local object. The constructor for each object is called when execution reaches the point where that object is defined. When execution reaches the end of `main`, the destructors for objects `fourth` then `second` are called in the reverse of their constructors' order. Object `third` is `static`, so it exists until program termination. The destructor for object `third` is called before the destructor for global object `first`, but after non-`static` local objects are destroyed.

Function `create` (lines 24–30) defines three objects—`fifth` (line 26) and `seventh` (line 28) are local automatic objects, and `sixth` (line 27) is a `static` local object. When `create` terminates, the destructors for objects `seventh` then `fifth` are called in the reverse of their constructors' order. Because `sixth` is `static`, it exists until program termination. The destructor for `sixth` is called before the destructors for `third` and `first`, but after all other non-`static` objects are destroyed. As an exercise, modify this program to call `create` twice—you'll see that the `static` object `sixth`'s constructor is called once when `create` is called the first time.

9.14 `Time` Class Case Study: A Subtle Trap —Returning a Reference or a Pointer to a `private` Data Member

A reference to an object is an alias for the object's name, so it may be used on the left side of an assignment statement. In this context, the reference makes a perfectly acceptable *lvalue* to which you can assign a value.

A member function can return a reference to a `private` data member of that class. If the reference return type is declared `const`, as we did for the `getName` member function of our Account class earlier in this chapter, the reference is a nonmodifiable *lvalue* and cannot be used to modify the data. However, subtle errors can occur if the reference return type is not declared `const`.

The program of Figs. 9.16–9.18 is a mechanical example that uses a simplified `Time` class to demonstrate the risk of returning a reference to a `private` data member. Member function `badSetHour` (declared in Fig. 9.16 in line 12 and defined in Fig. 9.17 in lines 30–33) returns an `int&` to the `m_hour` data member. **Such a reference return makes the result of a call to member function `badSetHour` an alias for `private` data member `hour`!** The function call can be used like the `private` data member, including as an *lvalue* in an assignment statement. So, clients of the class can overwrite the class's `private` data at will! A similar problem would occur if the function returned a pointer to the `private` data.

```
// Fig. 9.16: Time.h
// Time class definition.
// Member functions defined in Time.cpp

// prevent multiple inclusions of header
#pragma once

class Time {
public:
   void setTime(int hour, int minute, int second);
   int getHour() const;
   int& badSetHour(int hour); // dangerous reference return
private:
   int m_hour{0};
   int m_minute{0};
   int m_second{0};
};
```

Fig. 9.16 `Time` class declaration.

```
// Fig. 9.17: Time.cpp
// Time class member-function definitions.
#include <stdexcept>
#include "Time.h" // include definition of class Time

// set new Time value using 24-hour time
void Time::setTime(int hour, int minute, int second) {
   // validate hour, minute and second
   if (hour < 0 || hour >= 24) {
      throw std::invalid_argument{"hour was out of range"};
   }

   if (minute < 0 || minute >= 60) {
      throw std::invalid_argument{"minute was out of range"};
   }

   if (second < 0 || second >= 60) {
      throw std::invalid_argument{"second was out of range"};
   }

   m_hour = hour;
   m_minute = minute;
   m_second = second;
}

// return hour value
int Time::getHour() const {return m_hour;}

// poor practice: returning a reference to a private data member
int& Time::badSetHour(int hour) {
   setTime(hour, m_minute, m_second);
   return m_hour; // dangerous reference return
}
```

Fig. 9.17 `Time` class member-function definitions.

Figure 9.18 declares `Time` object `t` (line 9) and reference `hourRef` (line 12), which we initialize with the reference returned by `t.badSetHour(20)`. Lines 14–15 display `hour-Ref`'s value to show that `hourRef` breaks the class's encapsulation. Statements in `main` should not have access to the `private` data in a `Time` object. Next, line 16 uses the `hourRef` to set `hour`'s value to the invalid value 30. Lines 17–18 call `getHour` to show that assigning to `hourRef` modified `t`'s `private` data. Line 22 uses the `badSetHour` function call as an *lvalue* and assigns the invalid value 74 to the reference the function returns. Lines 24–25 call `getHour` again to show that line 22 modified the `private` data in the `Time` object `t`.

```
1  // fig09_18.cpp
2  // public member function that
3  // returns a reference to private data.
4  #include <iostream>
5  #include <fmt/format.h>
6  #include "Time.h" // include definition of class Time
7
8  int main() {
9     Time t{}; // create Time object
10
11    // initialize hourRef with the reference returned by badSetHour
12    int& hourRef{t.badSetHour(20)}; // 20 is a valid hour
13
14    std::cout << fmt::format(
15       "Valid hour before modification: {}\n", hourRef);
16    hourRef = 30; // use hourRef to set invalid value in Time object t
17    std::cout << fmt::format(
18       "Invalid hour after modification: {}\n\n", t.getHour());
19
20    // Dangerous: Function call that returns a reference can be
21    // used as an lvalue! POOR PROGRAMMING PRACTICE!!!!!!!!
22    t.badSetHour(12) = 74; // assign another invalid value to hour
23
24    std::cout << "After using t.badSetHour(12) as an lvalue, "
25       << fmt::format("hour is: {}\n", t.getHour());
26 }
```

```
Valid hour before modification: 20
Invalid hour after modification: 30

After using t.badSetHour(12) as an lvalue, hour is: 74
```

Fig. 9.18 `public` member function that returns a reference to `private` data.

Returning a reference or a pointer to a `private` data member breaks the class's encapsulation, making the client code dependent on the class's data representation. There are cases where doing this is appropriate. We'll show an example of this when we build our custom `MyArray` class in Section 11.6.

9.15 Default Assignment Operator

The assignment operator (`=`) can assign an object to another object of the same type. The **default assignment operator**[30] generated by the compiler copies each data member of the right operand into the same data member in the left operand. Figures 9.19 and 9.20 define a `Date` class. Line 15 of Fig.

9.21 uses the default assignment operator to assign `Date` object `date1` to `Date` object `date2`. In this case, `date1`'s `m_year`, `m_month` and `m_day` members are assigned to `date2`'s `m_year`, `m_month` and `m_day` members, respectively.
30. This is actually the default copy assignment operator. In Chapter 11, we'll distinguish between the copy assignment operator and the move assignment operator.

```
1  // Fig. 9.19: Date.h
2  // Date class declaration. Member functions are defined in Date.cpp.
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5
6  // class Date definition
7  class Date {
8  public:
9     Date(int year, int month, int day);
10    std::string toString() const;
11 private:
12    int m_year;
13    int m_month;
14    int m_day;
15 };
```

Fig. 9.19 `Date` class declaration.

```
1  // Fig. 9.20: Date.cpp
2  // Date class member-function definitions.
3  #include <fmt/format.h>
4  #include <string>
5  #include "Date.h" // include definition of class Date from Date.h
6
7  // Date constructor (should do range checking)
8  Date::Date(int year, int month, int day)
9     : m_year{year}, m_month{month}, m_day{day} {}
10
11 // return string representation of a Date in the format yyyy-mm-dd
12 std::string Date::toString() const {
13    return fmt::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
14 }
```

Fig. 9.20 `Date` class member-function definitions.

```
1  // fig09_21.cpp
2  // Demonstrating that class objects can be assigned
3  // to each other using the default assignment operator.
4  #include <fmt/format.h>
5  #include <iostream>
6  #include "Date.h" // include definition of class Date from Date.h
7  using namespace std;
8
9  int main() {
10    const Date date1{2006, 7, 4};
```

```
11     Date date2{2022, 1, 1};
12
13     std::cout << fmt::format("date1: {}\ndate2: {}\n\n",
14        date1.toString(), date2.toString());
15        date2 = date1; // uses the default assignment operator
16     std::cout << fmt::format("After assignment, date2: {}\n",
17        date2.toString());
18  }
```

```
date1: 2006-07-04
date2: 2022-01-01

After assignment, date2: 2006-07-04
```

Fig. 9.21 Class objects can be assigned to each other using the default assignment operator.

Copy Constructors

Objects may be passed as function arguments and may be returned from functions. Such passing and returning are performed using pass-by-value by default—a copy of the object is passed or returned. In such cases, C++ creates a new object and uses a **copy constructor** to copy the original object's data into the new object. For each class we've shown so far, the compiler provides a default copy constructor that copies each member of the original object into the corresponding member of the new object.[31]

31. In Chapter 11, we'll discuss cases in which the compiler uses move constructors, rather than copy constructors.

9.16 `const` Objects and `const` Member Functions

Let's see how the principle of least privilege applies to objects. Some objects do not need to be modifiable, so you should declare them `const`. Any attempt to modify a `const` object results in a compilation error. The statement

```
const Time noon{12, 0, 0};
```

declares a `const Time` object `noon` and initializes it to 12 noon (12 PM). It's possible to instantiate `const` and non-`const` objects of the same class.

SE C++ disallows calling a member function on a `const` object unless that member function is declared `const`. So, declare as `const` any member function that does not modify the object on which it's called.

Err A constructor must be allowed to modify an object to initialize it. A destructor must be allowed to perform its termination housekeeping before an object's memory is reclaimed by the system. So, declaring a constructor or destructor `const` is a compilation error. The "`const`ness" of a `const` object is enforced throughout the object's lifetime after the object is constructed.

Using `const` and Non-`const` Member Functions

The program of Fig. 9.22 uses a copy of class `Time` from Figs. 9.10 and 9.11, but removes `const` from function `to12HourString`'s prototype and definition to force a compilation error. We create two `Time` objects—non-`const` object `wakeUp` (line 6) and `const` object `noon` (line 7). The program attempts to invoke non-`const` member functions `setHour` (line 11) and `to12HourString` (line 15) on

the `const` object `noon`. In each case, the compiler generates an error message. The program also illustrates the three other member-function-call combinations on objects:

- a non-`const` member function on a non-`const` object (line 10),
- a `const` member function on a non-`const` object (line 12) and
- a `const` member function on a `const` object (lines 13–14).

The error messages generated for non-`const` member functions called on a `const` object are shown in the output window. We added blank lines for readability.

```
1  // fig09_22.cpp
2  // const objects and const member functions.
3  #include "Time.h" // include Time class definition
4
5  int main() {
6     Time wakeUp{6, 45, 0}; // non-constant object
7     const Time noon{12, 0, 0}; // constexpr object
8
9                                 // OBJECT      MEMBER FUNCTION
10    wakeUp.setHour(18);         // non-const   non-const
11    noon.setHour(12);        // const      non-const
12    wakeUp.getHour();           // non-const   const
13    noon.getMinute();           // const       const
14    noon.to24HourString();      // const       const
15    noon.to12HourString();   // const      non-const
16 }
```

clang++ compiler error messages:

```
fig09_22.cpp:11:4: error: 'this' argument to member function 'setHour' has
type 'const Time', but function is not marked const
   noon.setHour(12);         // const       non-const
   ^~~~

./Time.h:15:9: note: 'setHour' declared here
   void setHour(int hour); // set hour (after validation)
        ^

fig09_22.cpp:15:4: error: 'this' argument to member function 'to12Hour-
String' has type 'const Time', but function is not marked const
   noon.to12HourString();    // const       non-const
   ^~~~

./Time.h:25:16: note: 'to12HourString' declared here
   std::string to12HourString(); // 12-hour time format string
               ^
2 errors generated.
```

Fig. 9.22 `const` objects and `const` member functions.

SE A constructor must be a non-`const` member function, but it can still initialize a `const` object (Fig. 9.22, line 7). Recall from Fig. 9.11 that the `Time` constructor's definition calls non-`const` member function `setTime` to initialize a `Time` object. Invoking a non-`const` member function from the constructor for a `const` object is allowed. Again, the object is not `const` until the constructor finishes initializing the object.

Line 15 in Fig. 9.22 generates a compilation error even though `Time`'s member function `to12HourString` does not modify the object on which it's called. This fact is not sufficient—you must explicitly declare the function `const` for this call to be allowed by the compiler.

9.17 Composition: Objects as Members of Classes

SE An `AlarmClock` object needs to know when it's supposed to sound its alarm, so why not include a `Time` object as a member of the `AlarmClock` class? Such a software-reuse capability is called **composition** (or **aggregation**) and is sometimes referred to as a **has-a relationship**—a class can have objects of other classes as members.[32] You've already used composition in this chapter's `Account` class examples. Class `Account` contained a `string` object as a data member.

32. As you'll see in Chapter 10, classes also may be derived from other classes that provide attributes and behaviors the new classes can use—this is called inheritance.

You've seen how to pass arguments to a constructor. Now, let's see how a class's constructor can pass arguments to member-object constructors via member initializers. The next program uses classes `Date` (Figs. 9.23 and 9.24) and `Employee` (Figs. 9.25 and 9.26) to demonstrate composition. Class `Employee`'s definition (Fig. 9.25) has `private` data members `m_firstName`, `m_lastName`, `m_birthDate` and `m_hireDate`. Members `m_birth-Date` and `m_hireDate` are objects of class `Date`, which has `private` data members `m_year`, `m_month` and `m_day`.

```
1  // Fig. 9.23: Date.h
2  // Date class definition; member functions defined in Date.cpp
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5
6  class Date {
7  public:
8     static const int monthsPerYear{12}; // months in a year
9     Date(int year, int month, int day);
10    std::string toString() const; // date string in yyyy-mm-dd format
11    ~Date(); // implementation displays when destruction occurs
12 private:
13    int m_year; // any year
14    int m_month; // 1-12 (January-December)
15    int m_day; // 1-31 based on month
16
17    // utility function to check if day is proper for month and year
18    bool checkDay(int day) const;
19 };
```

Fig. 9.23 `Date` class definition.

```
1  // Fig. 9.24: Date.cpp
2  // Date class member-function definitions.
3  #include <array>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <stdexcept>
7  #include "Date.h" // include Date class definition
```

```
 8
 9  // constructor confirms proper value for month; calls
10  // utility function checkDay to confirm proper value for day
11  Date::Date(int year, int month, int day)
12     : m_year{year}, m_month{month}, m_day{day} {
13     if (m_month < 1 || m_month > monthsPerYear) { // validate the month
14        throw std::invalid_argument{"month must be 1-12"};
15     }
16
17     if (!checkDay(day)) { // validate the day
18        throw std::invalid_argument{
19           "Invalid day for current month and year"};
20     }
21
22     // output Date object to show when its constructor is called
23     std::cout << fmt::format("Date object constructor: {}\n", toString());
24  }
25
26  // gets string representation of a Date in the form yyyy-mm-dd
27  std::string Date::toString() const {
28     return fmt::format("{}-{:02d}-{:02d}", m_year, m_month, m_day);
29  }
30
31  // output Date object to show when its destructor is called
32  Date::~Date() {
33     std::cout << fmt::format("Date object destructor: {}\n", toString());
34  }
35
36  // utility function to confirm proper day value based on
37  // month and year; handles leap years, too
38  bool Date::checkDay(int day) const {
39     // we ignore element 0
40     static const std::array daysPerMonth{
41        0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
42
43     // determine whether testDay is valid for specified month
44     if (1 <= day && day <= daysPerMonth.at(m_month)) {
45        return true;
46     }
47
48     // February 29 check for leap year
49     if (m_month == 2 && day == 29 && (m_year % 400 == 0 ||
50        (m_year % 4 == 0 && m_year % 100 != 0))) {
51        return true;
52     }
54     return false; // invalid day, based on current m_month and m_year
55  }
```

Fig. 9.24 `Date` class member-function definitions.

```
1  // Fig. 9.25: Employee.h
2  // Employee class definition showing composition.
3  // Member functions defined in Employee.cpp.
4  #pragma once // prevent multiple inclusions of header
5  #include <string>
```

```
 6  #include <string_view>
 7  #include "Date.h" // include Date class definition
 8
 9  class Employee {
10  public:
11     Employee(std::string_view firstName, std::string_view lastName,
12        const Date& birthDate, const Date& hireDate);
13     std::string toString() const;
14     ~Employee(); // provided to confirm destruction order
15  private:
16     std::string m_firstName; // composition: member object
17     std::string m_lastName; // composition: member object
18     Date m_birthDate; // composition: member object
19     Date m_hireDate; // composition: member object
20  };
```

Fig. 9.25 `Employee` class definition showing composition.

```
 1  // Fig. 9.26: Employee.cpp
 2  // Employee class member-function definitions.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5  #include "Employee.h" // Employee class definition
 6  using namespace std;
 7
 8  // constructor uses member initializer list to pass initializer
 9  // values to constructors of member objects
10  Employee::Employee(std::string_view firstName, std::string_view lastName,
11     const Date &birthDate, const Date &hireDate)
12     : m_firstName{firstName}, m_lastName{lastName},
13       m_birthDate{birthDate}, m_hireDate{hireDate} {
14     // output Employee object to show when constructor is called
15     std::cout << fmt::format("Employee object constructor: {} {}\n",
16        m_firstName, m_lastName);
17  }
18
19  // gets string representation of an Employee object
20  std::string Employee::toString() const {
21     return fmt::format("{}, {} Hired: {} Birthday: {}", m_lastName,
22        m_firstName, m_hireDate.toString(), m_birthDate.toString());
23  }
24
25  // output Employee object to show when its destructor is called
26  Employee::~Employee() {
27     cout << fmt::format("Employee object destructor: {}, {}\n",
28        m_lastName, m_firstName);
29  }
```

Fig. 9.26 `Employee` class member-function definitions.

`Employee` Constructor's Member-Initializer List

SE CG The `Employee` constructor's prototype (Fig. 9.25, lines 11–12) specifies that the constructor has four parameters (`firstName`, `lastName`, `birthDate` and `hireDate`). In the constructor's definition (Fig. 9.26, lines 12–13), the first two parameters are passed via member initializers to

the `string` constructor for data members `firstName` and `lastName`. The last two are passed via member initializers to class `Date`'s constructor for data members `birthDate` and `hireDate`. The order of the member initializers does not matter. Data members are constructed in the order that they're declared in class `Employee`, not in the order they appear in the member-initializer list. For clarity, the C++ Core Guidelines recommend listing the member initializers in the order they're declared in the class.[33]

33. C++ Core Guidelines, "C.47: Define and Initialize Member Variables in the Order of Member Declaration." Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-order.

`Date` Class's Default Copy Constructor

As you study class `Date` (Fig. 9.23), notice it does not provide a constructor with a `Date` parameter. So, why can the `Employee` constructor's member-initializer brace initialize the `m_birthDate` and `m_hireDate` objects by passing `Date` objects to their constructors? As mentioned in Section 9.15, the compiler provides each class with a default copy constructor that copies each data member of the constructor's argument into the corresponding member of the object being initialized. Chapter 11 discusses how to define customized copy constructors.

Testing Classes `Date` and `Employee`

Figure 9.27 creates two `Date` objects (lines 9–10), then passes them as arguments to the constructor of the `Employee` object created in line 11. When an `Employee` is created, its constructor makes

- two calls to the `string` class's constructor (line 12 of Fig. 9.26) and
- two calls to the `Date` class's default copy constructor (line 13 of Fig. 9.26).

Line 13 of Fig. 9.27 displays the `Employee`'s data to show that it was initialized correctly.

```
 1  // fig09_27.cpp
 2  // Demonstrating composition--an object with member objects.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5  #include "Date.h" // Date class definition
 6  #include "Employee.h" // Employee class definition
 7
 8  int main() {
 9     const Date birth{1987 ,7, 24};
10     const Date hire{2018, 3, 12};
11     const Employee manager{"Sue", "Green", birth, hire};
12
13     std::cout << fmt::format("\n{}\n\n", manager.toString());
14  }
```

```
Date object constructor: 1987-07-24
Date object constructor: 2018-03-12
Employee object constructor: Sue Green

Green, Sue Hired: 2018-03-12 Birthday: 1987-07-24

Employee object destructor: Green, Sue
Date object destructor: 2018-03-12
Date object destructor: 1987-07-24
Date object destructor: 2018-03-12
```

```
Date object destructor: 1987-07-24
```

Fig. 9.27 Demonstrating composition—an object with member objects.

When each `Date` object is created in lines 9–10, the `Date` constructor (lines 11–24 of Fig. 9.24) displays a line of output to show that the constructor was called (see the first two lines of the sample output). However, line 11 of Fig. 9.27 causes two `Date` copy-constructor calls (line 13 of Fig. 9.26) that do not appear in this program's output. Since the compiler defines our `Date` class's copy constructor, it does not contain any output statements to demonstrate when it's called.

Class `Date` and class `Employee` each include a destructor (lines 32–34 of Fig. 9.24 and lines 26–29 of Fig. 9.26, respectively) that prints a message when an object of its class is destructed. The destructors help us show that objects are destructed from the outside in. The `Date` member objects are destructed after the enclosing `Employee` object.

Notice the last four lines in the output of Fig. 9.27. The last two lines are the outputs of the `Date` destructor running on `Date` objects `hire` (Fig. 9.27, line 10) and `birth` (line 9), respectively. The outputs confirm that the three objects created in `main` are destructed in the reverse order of their construction. The `Employee` destructor output is five lines from the bottom. The fourth and third lines from the bottom of the output show the destructors running for the `Employee`'s member objects `m_hireDate` (Fig. 9.25, line 19) and `m_birthDate` (line 18).

These outputs confirm that the `Employee` object is destructed from the outside in. The `Employee` destructor runs first (see the output five lines from the bottom). Then the member objects are destructed in the reverse order from which they were constructed. Class `string`'s destructor does not contain output statements, so we do not see the `firstName` and `lastName` objects being destructed.

What Happens When You Do Not Use the Member-Initializer List?

Err Perf If you do not initialize a member object explicitly, its default constructor will be called implicitly to initialize it. If there is no default constructor, a compilation error occurs. Values set by the default constructor can be changed later by *set* functions. However, this approach may require significant additional work and time for complex initialization. Initializing member objects via member initializers eliminates the overhead of "doubly initializing" member objects.

9.18 `friend` Functions and `friend` Classes

A **`friend` function** has access to a class's `public` and non-`public` members. A class may have as `friend`s

- stand-alone functions,
- entire classes (and thus all their functions) or
- specific member functions of other classes.

This section presents a mechanical example of how a `friend` function works. In Chapter 11, Operator Overloading, Copy/Move Semantics and Smart Pointers, we'll show `friend` functions that overload operators for use with objects of custom classes. You'll see that sometimes a member function cannot be used to define certain overloaded operators.

Declaring a `friend`

To declare a non-member function as a `friend` of a class, place the function prototype in the class definition and precede it with the keyword `friend`. To declare all member functions of an existing class `ClassTwo` as friends of class `ClassOne`, place in `ClassOne`'s definition a declaration of the form:

```
friend class ClassTwo;
```

Friendship Rules

SE These are the basic friendship rules:

- **Friendship is granted, not taken**—For class B to be a `friend` of class A, class A must declare that class B is its `friend`.
- **Friendship is not symmetric**—If class A is a `friend` of class B, you cannot infer that class B is a `friend` of class A.
- **Friendship is not transitive**—If class A is a `friend` of class B and class B is a `friend` of class C, you cannot infer that class A is a `friend` of class C.

`friend`s Are Not Subject to Access Modifiers

Member access notions of `public`, `protected` (Chapter 10) and `private` do not apply to `friend` declarations, so `friend` declarations can be placed anywhere in a class definition. We prefer to place `friend` declarations first inside the class definition's body and not precede them with any access specifier.

Modifying a Class's `private` Data with a `friend` Function

Figure 9.28 defines the free function `modifyX` as a `friend` of class `Count` (line 9), so `modifyX` can set `Count`'s `private` data member `m_x`.

```
1  // fig09_28.cpp
2  // Friends can access private members of a class.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include "fmt/format.h" // In C++20, this will be #include <format>
6
7  // Count class definition
8  class Count {
9     friend void setX(Count& c, int value); // friend declaration
10 public:
11    int getX() const {return m_x;}
12 private:
13    int m_x{0};
14 };
15
16 // function modifyX can modify private data of Count
17 // because modifyX is declared as a friend of Count (line 8)
18 void modifyX(Count& c, int value) {
19    c.m_x = value; // allowed because modifyX is a friend of Count
20 }
21
22 int main() {
23    Count counter{}; // create Count object
24
25    std::cout << fmt::format("Initial counter.m_x: {}\n", counter.getX());
26    modifyX(counter, 8); // change x's value using a friend function
27    std::cout << fmt::format("counter.m_x after modifyX: {}\n",
```

```
28         counter.getX());
29   }
Initial counter.m_x: 0
counter.m_x after modifyX: 8
```

Fig. 9.28 `Friend`s can access `private` members of a class.

Function `modifyX` (lines 18–20) is a stand-alone (free) function, not a `Count` member function. So, when we call `modifyX` in `main` to modify the `Count` object `counter` (line 26), we must pass `counter` as an argument to `modifyX`. Function `modifyX` can access class `Count`'s `private` data member `m_x` (line 19) only because the function was declared as a `friend` of class `Count` (line 9). If you remove the `friend` declaration, you'll receive error messages indicating that function `modifyX` cannot access class `Count`'s `private` data member `m_x`.

9.19 The `this` Pointer

There's only one copy of each class's functionality, but there can be many objects of a class. So, how do member functions know which object's data members to manipulate? Every object's member functions access the object through a pointer called **`this`** (a C++ keyword), which is initialized with an implicit argument passed to each of the object's non-`static`[34] member functions.

34. Section 9.20 introduces `static` class members and explains why the `this` pointer is not implicitly passed to `static` member functions.

Using the `this` Pointer to Avoid Naming Collisions

SE Member functions use the `this` pointer implicitly (as we've done so far) or explicitly to reference an object's data members and other member functions. One explicit use of the `this` pointer is to avoid naming conflicts between a class's data members and constructor or member-function parameters. If a member function uses a local variable and a data member with the same name, the local variable is said to **hide** or **shadow** the data member. Using just the variable name in the member function's body refers to the local variable rather than the data member.

You can access the data member explicitly by qualifying its name with `this->`. For instance, if class `Account` had a `string` data member `name`, we could implement its `set-Name` function as follows:

```
void Account::setName(std::string_view name) {
   this->name = name; // use this-> to access data member
}
```

where `this->name` represents a data member called `name`. You can avoid such naming collisions by naming your data members with the `"m_"` prefix, as shown in our classes so far.

Type of the `this` Pointer

The `this` pointer's type depends on the object's type and whether the member function in which `this` is used is declared `const`:

- In a non-`const` member function of class `Time`, the `this` pointer is a `Time*`—a pointer to a `Time` object.
- In a `const` member function, `this` is a `const Time*`—a pointer to a `Time` constant.

9.19.1 Implicitly and Explicitly Using the `this` Pointer to Access an Object's Data Members

Figure 9.29 is a mechanical example that demonstrates implicit and explicit use of the `this` pointer in a member function to display the `private` data `m_x` of a `Test` object. In Section 9.19.2 and in Chapter 11, we show some substantial and subtle examples of using `this`.

```
 1  // fig09_29.cpp
 2  // Using the this pointer to refer to object members.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5
 6  class Test {
 7  public:
 8     explicit Test(int value);
 9     void print() const;
10  private:
11     int m_x{0};
12  };
13
14  // constructor
15  Test::Test(int value) : m_x{value} {} // initialize m_x to value
16
17  // print m_x using implicit then explicit this pointers;
18  // the parentheses around *this are required due to precedence
19  void Test::print() const {
20     // implicitly use the this pointer to access the member m_x
21     std::cout << fmt::format("        m_x = {}\n", m_x);
22
23     // explicitly use the this pointer and the arrow operator
24     // to access the member m_x
25     std::cout << fmt::format("  this->m_x = {}\n", this->m_x);
26
27     // explicitly use the dereferenced this pointer and
28     // the dot operator to access the member m_x
29     std::cout << fmt::format("(*this).m_x = {}\n", (*this).m_x);
30  }
31
32  int main() {
33     const Test testObject{12}; // instantiate and initialize testObject
34     testObject.print();
35  }
```

```
        x = 12
  this->x = 12
(*this).x = 12
```

Fig. 9.29 Using the `this` pointer to refer to object members.

For illustration purposes, member function `print` (lines 19–30) first displays `m_x` using the `this` pointer implicitly (line 21)—only the data member's name is specified. Then `print` uses two different notations to access `m_x` through the `this` pointer:

- `this->m_x` (line 25) and
- `(*this).m_x` (line 29).

Err The parentheses around `*this` (line 29) are required because the dot operator (`.`) has higher precedence than the `*` pointer-dereferencing operator. Without the parentheses, the

expression `*this.m_x` would be evaluated as `*(this.m_x)`. The dot operator cannot be used with a pointer, so this would be a compilation error.

9.19.2 Using the `this` Pointer to Enable Cascaded Function Calls

SE Another use of the `this` pointer is to enable **cascaded member-function calls**—that is, invoking multiple functions sequentially in the same statement, as you'll see in line 11 of Fig. 9.32. The program of Figs. 9.30–9.32 modifies class `Time`'s `setTime`, `setHour`, `set-Minute` and `setSecond` functions such that each returns a reference to the `Time` object on which it's called. This reference enables cascaded member-function calls. In Fig. 9.31, the last statement in `setTime`'s body returns a reference to `*this` (line 30). Functions `setHour`, `setMinute` and `setSecond` each call `setTime` and return its `Time&` result.

```
1  // Fig. 9.30: Time.h
2  // Time class modified to enable cascaded member-function calls.
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5
6  class Time {
7  public:
8     // default constructor because it can be called with no arguments
9     explicit Time(int hour = 0, int minute = 0, int second = 0);
10
11    // set functions
12    Time& setTime(int hour, int minute, int second);
13    Time& setHour(int hour); // set hour (after validation)
14    Time& setMinute(int minute); // set minute (after validation)
15    Time& setSecond(int second); // set second (after validation)
16
17    int getHour() const; // return hour
18    int getMinute() const; // return minute
19    int getSecond() const; // return second
20    std::string to24HourString() const; // 24-hour time format string
21    std::string to12HourString() const; // 12-hour time format string
22 private:
23    int m_hour{0}; // 0 - 23 (24-hour clock format)
24    int m_minute{0}; // 0 - 59
25    int m_second{0}; // 0 - 59
26 };
```

Fig. 9.30 `Time` class modified to enable cascaded member-function calls.

```
1  // Fig. 9.31: Time.cpp
2  // Time class member-function definitions.
3  #include <fmt/format.h>
4  #include <stdexcept>
5  #include "Time.h" // Time class definition
6
7  // Time constructor initializes each data member
8  Time::Time(int hour, int minute, int second) {
9     setTime(hour, minute, second);
10 }
```

```
11
12 // set new Time value using 24-hour time
13 Time& Time::setTime(int hour, int minute, int second) {
14    // validate hour, minute and second
15    if (hour < 0 || hour >= 24) {
16       throw std::invalid_argument{"hour was out of range"};
17    }
18
19    if (minute < 0 || minute >= 60) {
20       throw std::invalid_argument{"minute was out of range"};
21    }
22
23    if (second < 0 || second >= 60) {
24       throw std::invalid_argument{"second was out of range"};
25    }
26
27    m_hour = hour;
28    m_minute = minute;
29    m_second = second;
30    return *this; // enables cascading
31 }
32
33 // set hour value
34 Time& Time::setHour(int hour) {
35    return setTime(hour, m_minute, m_second);
36 }
37
38 // set minute value
39 Time& Time::setMinute(int minute) {
40    return setTime(m_hour, minute, m_second);
41 }
42
43 // set second value
44 Time& Time::setSecond(int second) {
45    return setTime(m_hour, m_minute, second);
46 }
47
48 // get hour value
49 int Time::getHour() const {return m_hour;}
50
51 // get minute value
52 int Time::getMinute() const {return m_minute;}
53
54 // get second value
55 int Time::getSecond() const {return m_second;}
56
57 // return Time as a string in 24-hour format (HH:MM:SS)
58 std::string Time::to24HourString() const {
59    return fmt::format("{:02d}:{:02d}:{:02d}",
60       getHour(), getMinute(), getSecond());
61 }

62
63 // return Time as string in 12-hour format (HH:MM:SS AM or PM)
```

```
64  std::string Time::to12HourString() const {
65     return fmt::format("{}:{:02d}:{:02d} {}",
66        ((getHour() % 12 == 0) ? 12 : getHour() % 12),
67        getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
68  }
```

Fig. 9.31 `Time` class member-function definitions modified to enable cascaded member-function calls.

In Fig. 9.32, we create `Time` object `t` (line 9), then use it in cascaded member-function calls (lines 11 and 19).

```
 1  // fig09_32.cpp
 2  // Cascading member-function calls with the this pointer.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5  #include "Time.h" // Time class definition
 6  using namespace std;
 7
 8  int main() {
 9     Time t{}; // create Time object
10
11     t.setHour(18).setMinute(30).setSecond(22); // cascaded function calls
12
13     // output time in 24-hour and 12-hour formats
14     std::cout << fmt::format("24-hour time: {}\n12-hour time: {}\n\n",
15        t.to24HourString(), t.to12HourString());
16
17     // cascaded function calls
18     std::cout << fmt::format("New 12-hour time: {}\n",
19        t.setTime(20, 20, 20).to12HourString());
20  }
```

```
24-hour time: 18:30:22
12-hour time: 6:30:22 PM

New 12-hour time: 8:20:20 PM
```

Fig. 9.32 Cascading member-function calls with the `this` pointer.

Why does the technique of returning `*this` as a reference work? The dot operator (`.`) groups left-to-right, so line 11

```
t.setHour(18).setMinute(30).setSecond(22);
```

first evaluates `t.setHour(18)`, which returns a reference to (the updated) object `t` as the value of this function call. The remaining expression is then interpreted as

```
t.setMinute(30).setSecond(22);
```

The `t.setMinute(30)` call executes and returns a reference to the (further updated) object `t`. The remaining expression is interpreted as

```
t.setSecond(22);
```

Line 19 (Fig. 9.32) also uses cascading.[35] We cannot chain another `Time` member-function call after `to12HourString`, because it does not return a `Time&`. However, we could chain a call to a `string` member function because `to12HourString` returns a `string`. Chapter 11 presents practical examples of cascaded function calls, such as in `cout` statements with the `<<` operator and in `cin` statements with the `>>` operator.

35. This particular chained call is for demonstration purposes only. Function calls that modify objects should be placed in stand-alone statements to avoid order-of-evaluation issues.

9.20 `static` Class Members: Classwide Data and Member Functions

There is an exception to the rule that each object has its own copy of its class's data members. In some cases, all objects of a class should share only one copy of a variable. A **`static` data member** is used for these and other reasons. Such a variable represents "classwide" information—that is, data shared by all objects of the class. You can use `static` data members to save storage when all objects of a class can share a single copy of the data.

Motivating Classwide Data

Let's further explore the need for `static` classwide data with an example. Suppose that we have a video game with `Martian`s and other space creatures. Each `Martian` tends to be brave and willing to attack other space creatures when the `Martian` is aware that at least five `Martian`s are present. If fewer than five are present, each `Martian` becomes cowardly. So each `Martian` needs to know the `martianCount`. We could endow each object of class `Martian` with `martianCount` as a data member. If we do, every `Martian` will have its own copy of the data member. Every time we create a new `Martian`, we'd have to update the data member `martianCount` in all `Martian` objects. Doing this would require every `Martian` object to know about all other `Martian` objects in memory. This wastes space with redundant `martianCount` copies and wastes time updating the separate copies. Instead, we declare `martianCount` to be `static` to make it classwide data. Every `Martian` can access `martianCount` as if it were a data member, but only one copy of the `static martianCount` is maintained in the program. This saves space. We have the `Martian` constructor increment `static` variable `martianCount` and the `Martian` destructor decrement `martian-Count`. Because there's only one copy, we do not have to increment or decrement separate copies of `martianCount` for every `Martian` object.

Scope and Initialization of `static` Data Members

17 A class's `static` data members have class scope. A `static` data member must be initialized exactly once. Fundamental-type `static` data members are initialized by default to `0`. A `static const` data member can have an in-class initializer. As of C++17, you also may use in-class initializers for a non-`const static` data member by preceding its declaration with the `inline` keyword (as you'll see in Fig. 9.33). If a `static` data member is an object of a class that provides a default constructor, the `static` data member need not be explicitly initialized because its default constructor will be called.

Accessing `static` Data Members

A class's `static` members exist even when no objects of that class exist. To access a `public static` class data member or member function, simply prefix the class name and the scope resolution operator (`::`) to the member name. For example, if our `martianCount` variable is `public`, it can be accessed with `Martian::martianCount`, even when there are no `Martian` objects.

SE A class's `private` (and `protected`; Chapter 10) `static` members are normally accessed through the class's `public` member functions or `friend`s. To access a `private static` or `protected static` data member when no objects of the class exist, provide a `public static` **member function** and call the function by prefixing its name with the class name and scope resolution operator. A `static` member function is a service of the class as a whole, not of a specific object of the class.

Demonstrating `static` Data Members

This example demonstrates a `private inline static` data member called `m_count`, which is initialized to `0` (Fig. 9.33, line 22), and a `public static` member function called `get-Count` (Fig. 9.33, line 16).

A `static` data member also can be initialized at file scope in the class's implementation file. For instance, we could have placed the following statement in `Employee.cpp` (Fig. 9.34) after the `Employee.h` header is included:

```
int Employee::count{0};
```

```
1  // Fig. 9.33: Employee.h
2  // Employee class definition with a static data member to
3  // track the number of Employee objects in memory
4  #pragma once
5  #include <string>
6  #include <string_view>
7
8  class Employee {
9  public:
10    Employee(std::string_view firstName, std::string_view lastName);
11    ~Employee(); // destructor
12    const std::string& getFirstName() const; // return first name
13    const std::string& getLastName() const; // return last name
14
15    // static member function
16    static int getCount(); // return # of objects instantiated
17 private:
18    std::string m_firstName;
19    std::string m_lastName;
20
21    // static data
22    inline static int m_count{0}; // number of objects instantiated
23 };
```

Fig. 9.33 `Employee` class definition with a `static` data member to track the number of `Employee` objects in memory.

In Fig. 9.34, line 10 defines `static` member function `getCount`—note this does not include the `static` keyword, which cannot be applied to a member definition that appears outside the class definition. In this program, data member `m_count` maintains a count of the number of `Employee` objects in memory at a given time. When `Employee` objects exist, member `m_count` can be referenced through any member function of an `Employee` object, as shown in the constructor (line 16) and the destructor (line 25).

```
1  // Fig. 9.34: Employee.cpp
2  // Employee class member-function definitions.
3  #include <fmt/format.h>
```

```
 4  #include <iostream>
 5  #include "Employee.h" // Employee class definition
 6  using namespace std;
 7
 8  // define static member function that returns number of
 9  // Employee objects instantiated (declared static in Employee.h)
10  int Employee::getCount() {return m_count;}
11
12  // constructor initializes non-static data members and
13  // increments static data member count
14  Employee::Employee(string_view firstName, string_view lastName)
15     : m_firstName(firstName), m_lastName(lastName) {
16     ++m_count; // increment static count of employees
17     std::cout << fmt::format("Employee constructor called for {} {}\n",
18        m_firstName, m_lastName);
19  }
20
21  // destructor decrements the count
22  Employee::~Employee() {
23     std::cout << fmt::format("~Employee() called for {} {}\n",
24        m_firstName, m_lastName);
25     --m_count; // decrement static count of employees
26  }
27
28  // return first name of employee
29  const string& Employee::getFirstName() const {return m_firstName;}
30
31  // return last name of employee
32  const string& Employee::getLastName() const {return m_lastName;}
```

Fig. 9.34 `Employee` class member-function definitions.

Figure 9.35 uses `static` member function `getCount` to determine the number of `Employee` objects in memory at various points in the program. The program calls `Employee::getCount()`:

- before any `Employee` objects have been created (line 11),
- after two `Employee` objects have been created (line 23) and
- after those `Employee` objects have been destroyed (line 34).

When `Employee` objects are in scope, you also can call `getCount` on those objects. For example, in line 23, we could have written either

```
e1.getCount()
```

or

```
e2.getCount()
```

Either would return the current value of class `Employee`'s `static m_count`.

```
 1  // fig09_35.cpp
 2  // static data member tracking the number of objects of a class.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5  #include "Employee.h" // Employee class definition
 6
```

```
 7  int main() {
 8     // no objects exist; use class name and scope resolution
 9     // operator to access static member function getCount
10     std::cout << fmt::format("Initial employee count: {}\n",
11        Employee::getCount()); // use class name
12
13     // the following scope creates and destroys
14     // Employee objects before main terminates
15     {
16        const Employee e1{"Susan", "Baker"};
17        const Employee e2{"Robert", "Jones"};
18
19        // two objects exist; call static member function getCount again
20        // using the class name and the scope resolution operator
21        std::cout << fmt::format(
22           "Employee count after creating objects: {}\n\n",
23           Employee::getCount());
24
25        std::cout << fmt::format("Employee 1: {} {}\nEmployee 2: {} {}\n\n",
26           e1.getFirstName(), e1.getLastName(),
27           e2.getFirstName(), e2.getLastName());
28  }
29
30  // no objects exist, so call static member function getCount again
31  // using the class name and the scope resolution operator
32  std::cout << fmt::format(
33     "Employee count after objects are deleted: {}\n",
34     Employee::getCount());
35  }
```

```
Initial employee count: 0
Employee constructor called for Susan Baker
Employee constructor called for Robert Jones
Employee count after creating objects: 2

Employee 1: Susan Baker
Employee 2: Robert Jones

~Employee() called for Robert Jones
~Employee() called for Susan Baker
Employee count after objects are deleted: 0
```

Fig. 9.35 `static` data member tracking the number of objects of a class.

Lines 15–28 in `main` define a nested scope. Recall that local variables exist until the scope in which they're defined terminates. In this example, we create two `Employee` objects in the nested scope (lines 16–17). As each constructor executes, it increments class `Employee`'s `static` data member `count`. These `Employee` objects are destroyed when the program reaches line 28. At that point, each object's destructor executes and decrements class `Employee`'s `static` data member `count`.

`static` Member Function Notes

SE Err A member function should be declared `static` if it does not access the class's non-`static` data members or non-`static` member functions. A `static` member function does not have a `this` pointer because `static` data members and `static` member functions exist independently of any objects of a class. The `this` pointer must refer to a specific object, but a `static` member function can be

called when there are no objects of its class in memory. So, using the `this` pointer in a `static` member function is a compilation error.

Err A `static` member function may not be declared `const`. The `const` qualifier indicates that a function cannot modify the contents of the object on which it operates, but `static` member functions exist and operate independently of any objects of the class. So, declaring a `static` member function `const` is a compilation error.

20 9.21 Aggregates in C++20

Section 9.4.1 of the C++ standard document

```
http://wg21.link/n4861
```

describes an **aggregate type** as a built-in array, an `array` object or an object of a class that

- does not have user-declared constructors,
- does not have `private` or `protected` (Chapter 10) non-`static` data members,
- does not have `virtual` functions (Chapter 10) and
- does not have `private` (Chapter 10), `protected` (Chapter 10) or `virtual` (online Chapter 20) base classes.

20 The requirement for no user-declared constructors was a C++20 change to prevent a case in which initializing an aggregate object could circumvent calling a user-declared constructor.[36]

36. "Prohibit Aggregates with User-Declared Constructors." Accessed January 6, 2022. http://wg21.link/p1008.

You can define an aggregate using a class in which all the data is `public`. However, **a `struct` is a class that contains only `public` members by default**. The following `struct` defines an aggregate type named `Record` containing four `public` data members:

```
struct Record {
   int account;
   string first;
   string last;
   double balance;
};
```

CG The C++ Core Guidelines recommend using `class` rather than `struct` if any data member or member function needs to be non-`public`.[37]

37. C++ Core Guidelines, "C.8: Use `class` Rather Than `struct` if Any Member Is Non-Public." Accessed January 6, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-class.

9.21.1 Initializing an Aggregate

You can initialize an object of aggregate type `Record` as follows:

```
Record record{100, "Brian", "Blue", 123.45};
```

11 In C++11, you could not use a braced initializer for an aggregate-type object if any of the type's non-`static` data-member declarations contained in-class initializers. For example, the initialization

above would have generated a compilation error if the aggregate type `Record` were defined with a default value for `balance`, as in:

```
struct Record {
   int account;
   std::string first;
   std::string last;
   double balance{0.0};
};
```

14 C++14 removed this restriction. Also, if you initialize an aggregate-type object with fewer initializers than there are data members in the object, as in

```
Record record{0, "Brian", "Blue"};
```

the remaining data members are initialized as follows:

- Data members with in-class initializers use those values—in the preceding case, `record`'s `balance` is set to `0.0`.
- Data members without in-class initializers are initialized with empty braces (`{}`). Empty-brace initialization sets fundamental-type variables to `0`, sets `bool`s to `false` and **value initializes** objects.

20 9.21.2 C++20: Designated Initializers

As of C++20, aggregates now support **designated initializers** in which you can specify which data members to initialize by name. Using the preceding `struct Record` definition, we could initialize a `Record` object as follows:

```
Record record{.first{"Sue"}, .last{"Green"}};
```

which explicitly initializes only a subset of the data members. Each explicitly named data member is preceded by a dot (`.`). The identifiers you specify must be listed in the same order as they're declared in the aggregate type. The preceding statement initializes the data members `first` and `last` to `"Sue"` and `"Green"`, respectively. The remaining data members get their default initializer values:

- `account` is set to `0` and
- `balance` is set to its default value in the type definition—in this case, `0.0`.

Other Benefits of Designated Initializers[38]

38. "Designated Initialization." Accessed July 12, 2020. http://wg21.link/p0329r0.

SE Adding new data members to an aggregate type will not break existing statements that use designated initializers. Any new data members that are not explicitly initialized simply receive their default initialization. Designated initializers also improve compatibility with the C programming language, which has had this feature since C99.

9.22 Objects-Natural Case Study: Serialization with JSON

More and more computing today is done “in the cloud”—that is, distributed across the Internet. Many applications you use daily communicate over the Internet with **cloud-based services** that use massive clusters of computing resources (computers, processors, memory, disk drives, databases, etc.).

A service that provides access to itself over the Internet is known as a **web service**. Applications typically communicate with web services by sending and receiving JSON objects. **JSON (JavaScript Object Notation)** is a text-based, human- and-computer-readable data-interchange format that represents objects as collections of name–value pairs. JSON has become the preferred data format for transmitting objects across platforms.

JSON Data Format

Each JSON object contains a comma-separated list of property names and values in curly braces. For example, the following name–value pairs might represent a client record:

```
{"account": 100, "name": "Jones", "balance": 24.98}
```

JSON also supports arrays as comma-separated values in square brackets. For example, the following represents a JSON array of numbers:

```
[100, 200, 300]
```

Values in JSON objects and arrays can be

- strings in double quotes (like `"Jones"`),
- numbers (like `100` or `24.98`),
- JSON Boolean values (represented as `true` or `false`),
- `null` (to represent no value),
- arrays of any valid JSON value and
- other JSON objects.

JSON arrays may contain elements of the same or different types.

Serialization

Converting an object into another format for storage locally or for transmission over the Internet is known as **serialization**. Similarly, reconstructing an object from serialized data is known as **deserialization**. JSON is just one of several serialization formats. Another common format is XML (eXtensible Markup Language).

Serialization Security

Sec Some programming languages have their own serialization mechanisms that use a language-native format. Deserializing objects using these native serialization formats is a source of various security issues. According to the Open Web Application Security Project (OWASP), these native mechanisms “can be repurposed for malicious effect when operating on untrusted data. Attacks

against deserializers have been found to allow denial-of-service, access control, and remote code execution (RCE) attacks."[39] OWASP also indicates that you can significantly reduce attack risk by avoiding language-native serialization formats in favor of "pure data" formats like JSON or XML.

39. "Deserialization Cheat Sheet." OWASP Cheat Sheet Series. Accessed July 18, 2020. https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html.

`cereal` Header-Only Serialization Library

The **`cereal` header-only library**[40] serializes objects to and deserializes objects from JSON (which we'll demonstrate), XML or binary formats. The library supports fundamental types and can handle most standard library types if you include each type's appropriate `cereal` header. As you'll see in the next section, `cereal` also supports custom types. The `cereal` documentation is available at

40. Copyright © 2017. W. Shane Grant and Randolph Voorhies, `cereal`—A C++11 library for serialization. http://uscilab.github.io/cereal/. All rights reserved.

```
https://uscilab.github.io/cereal/index.html
```

We've included the cereal library for your convenience in the `libraries` folder with the book's examples. You must point your IDE or compiler at the library's `include` folder, as you've done in several earlier Objects-Natural case studies.

9.22.1 Serializing a `vector` of Objects Containing `public` Data

Let's begin by serializing objects containing only `public` data. In Fig. 9.36, we

- create a `vector` of `Record` objects and display its contents,
- use `cereal` to serialize it to a text file, then
- deserialize the file's contents into a `vector` of `Record` objects and display the deserialized `Record`s.

To perform JSON serialization, include the `cereal` header `json.hpp` (line 3). To serialize a `std::vector`, include the `cereal` header `vector.hpp` (line 4).

```
1  // fig09_36.cpp
2  // Serializing and deserializing objects with the cereal library.
3  #include <cereal/archives/json.hpp>
4  #include <cereal/types/vector.hpp>
5  #include <fmt/format.h>
6  #include <fstream>
7  #include <iostream>
8  #include <vector>
```

Fig. 9.36 Serializing and deserializing objects with the `cereal` library.

Aggregate Type `Record`

`Record` is an aggregate type defined as a `struct`. Recall that an aggregate's data must be `public`, which is the default for a `struct` definition.

```
10 struct Record {
11    int account{};
```

```
12     std::string first{};
13     std::string last{};
14     double balance{};
15  };
```

Function `serialize` for `Record` Objects

The `cereal` library allows you to designate how to perform serialization several ways. If the types you wish to serialize have all `public` data, you can simply define a function template `serialize` (lines 19–25) that receives an `Archive` as its first parameter and an object of your type as the second.[41] This function is called both for serializing and deserializing the `Record` objects. Using a function template enables you to choose among serialization and deserialization using JSON, XML or binary formats by passing an object of the appropriate `cereal` archive type. The library provides archive implementations for each case.

41. Function `serialize` also may be defined as a member function template with only an `Archive` parameter. For details, see https://uscilab.github.io/cereal/serialization_functions.html.

```
17  // function template serialize is responsible for serializing and
18  // deserializing Record objects to/from the specified Archive
19  template <typename Archive>
20  void serialize(Archive& archive, Record& record) {
21     archive(cereal::make_nvp("account", record.account),
22        cereal::make_nvp("first", record.first),
23        cereal::make_nvp("last", record.last),
24        cereal::make_nvp("balance", record.balance));
25  }
```

Each `cereal` archive type has an overloaded parentheses operator that enables you to use an archive parameter as a function name, as in lines 21–24. Depending on whether you're serializing or deserializing a `Record`, this function will either

- output the contents of the `Record` to a specified stream or
- input previously serialized data from a specified stream and create a `Record` object.

Each call to `cereal::make_nvp` (that is, "make name–value pair"), like line 21

```
cereal::make_nvp("account", record.account)
```

is primarily for the serialization step. It makes a name–value pair with the name in the first argument (in this case, `"account"`) and the value in the second argument (in this case, the `int` value `record.account`). Naming the values is not required but makes the JSON output more readable, as you'll soon see. Otherwise, cereal uses names like `value0`, `value1`, etc.

Function `displayRecords`

We provide function `displayRecords` to show you the contents of our `Record` objects before serialization and after deserialization. The function simply displays the contents of each `Record` in the `vector` it receives as an argument:

```
27  // display record at command line
28  void displayRecords(const std::vector<Record>& records) {
```

```
29     for (const auto& r : records) {
30        std::cout << fmt::format("{} {} {} {:.2f}\n",
31           r.account, r.first, r.last, r.balance);
32     }
33  }
```

Creating `Record` Objects to Serialize

Lines 36–39 in `main` create a `vector` and initialize it with two `Record`s (lines 37 and 38). The compiler uses class template argument deduction (CTAD) to determine the `vector`'s element type from the `Record`s in the initializer list. Each `Record` is initialized with aggregate initialization. Line 42 outputs the `vector`'s contents to confirm that the two `Record`s were initialized properly.

```
35  int main() {
36     std::vector records{
37        Record{100, "Brian", "Blue", 123.45},
38        Record{200, "Sue", "Green", 987.65}
39     };
40
41     std::cout << "Records to serialize:\n";
42     displayRecords(records);
43
```

```
Records to serialize:
100 Brian Blue 123.45
200 Sue Green 987.65
```

Serializing `Record` Objects with `cereal::JSONOutputArchive`

A `cereal::JSONOutputArchive` serializes data in JSON format to a specified stream, such as the standard output stream or a stream representing a file. Line 45 attempts to open the file `records.json` for writing. If it is successful, line 46 creates a `cereal::JSONOutput-Archive` object named `archive` and initializes it with the `output ofstream` object so `archive` can write the JSON data into a file. Line 47 uses the `archive` object to output a name–value pair with the name `"records"` and the `vector` of `Record`s as its value. Part of serializing the `vector` is serializing each of its elements. So line 49 also results in one call to `serialize` (lines 19–25) for each `Record` object in the `vector`.

```
44    // serialize vector of Records to JSON and store in text file
45    if (std::ofstream output{"records.json"}) {
46       cereal::JSONOutputArchive archive{output};
47       archive(cereal::make_nvp("records", records)); // serialize records
48    }
```

Contents of `records.json`

After line 47 executes, the file `records.json` contains the following JSON data:

```
{
   "records": [
      {
         "account": 100,
         "first": "Brian",
         "last": "Blue",
         "balance": 123.45
      },
      {
         "account": 200,
         "first": "Sue",
         "last": "Green",
         "balance": 987.65
      }
   ]
}
```

The outer braces represent the entire JSON document. The darker box highlights the document's one name–value pair named `"records"`, which has as its value a JSON array containing the two JSON objects in the lighter boxes. The JSON array represents the `vector records` that we serialized in line 47. Each of the JSON objects in the array contains one `Record`'s four name–value pairs that were serialized by the `serialize` function.

Deserializing `Record` Objects with `cereal::JSONInputArchive`

Next, let's deserialize the data and use it to fill a separate `vector` of `Record` objects. For `cereal` to recreate objects in memory, it must have access to each type's default constructor. It will use that to create an object, then directly access that object's data members to place the data into the object. Like classes, the compiler provides a `public` default constructor for `struct`s if you do not define a custom constructor.

```
50      // deserialize JSON from text file into vector of Records
51      if (std::ifstream input{"records.json"}) {
52         cereal::JSONInputArchive archive{input};
53         std::vector<Record> deserializedRecords{};
54         archive(deserializedRecords); // deserialize records
55         std::cout << "\nDeserialized records:\n";
56         displayRecords(deserializedRecords);
57       }
58   }
```

```
Deserialized records:
100 Brian Blue 123.45
200 Sue Green 987.65
```

A **`cereal::JSONInputArchive`** deserializes JSON format from a specified stream. Line 51 attempts to open the file `records.json` for reading. If it is successful, line 54 creates the `cereal::JSONInputArchive` object `archive` and initializes it with the `input ifstream` object so `archive` can read JSON data from the `records.json` file. Line 53 creates an empty `vector` of `Record`s into which we'll read the JSON data. Line 54 uses the `archive` object to deserialize the file's data into the `deserializedRecords` object. Part of deserializing the `vector` is deserializing its elements. This again results in calls to `serialize` (lines 19–25), but because `archive` is a `cereal::JSONInputArchive`, each call to `serialize` reads one `Record`'s JSON data, creates a `Record` object and inserts the data into it.

9.22.2 Serializing a `vector` of Objects Containing `private` Data

It is also possible to serialize objects containing `private` data. To do so, **you must declare the `serialize` function as a `friend` of the class, so it can access the class's `private` data**. To demonstrate serializing `private` data, we created a copy of Fig. 9.36 and replaced the aggregate `Record` definition with the class `Record` in lines 12–35 of Fig. 9.37.

```
1  // fig09_37.cpp
2  // Serializing and deserializing objects containing private data.
3  #include <cereal/archives/json.hpp>
4  #include <cereal/types/vector.hpp>
5  #include <fmt/format.h>
6  #include <fstream>
7  #include <iostream>
8  #include <string>
9  #include <string_view>
10 #include <vector>
11
12 class Record {
13    // declare serialize as a friend for direct access to private data
14    template<typename Archive>
15    friend void serialize(Archive& archive, Record& record);
16
17 public:
18    // constructor
19    explicit Record(int account = 0, std::string_view first = "",
20       std::string_view last = "", double balance = 0.0)
21       : m_account{account}, m_first{first},
22         m_last{last}, m_balance{balance} {}
23
24    // get member functions
25    int getAccount() const {return m_account;}
26    const std::string& getFirst() const {return m_first;}
27    const std::string& getLast() const {return m_last;}
28    double getBalance() const {return m_balance;}
29
30 private:
31    int m_account{};
32    std::string m_first{};
33    std::string m_last{};
34    double m_balance{};
35 };
36
37 // function template serialize is responsible for serializing and
38 // deserializing Record objects to/from the specified Archive
39 template <typename Archive>
40 void serialize(Archive& archive, Record& record) {
41    archive(cereal::make_nvp("account", record.m_account),
42       cereal::make_nvp("first", record.m_first),
43       cereal::make_nvp("last", record.m_last),
44       cereal::make_nvp("balance", record.m_balance));
45 }
46
```

```
47  // display record at command line
48  void displayRecords(const std::vector<Record>& records) {
49     for (const auto& r : records) {
50        std::cout << fmt::format("{} {} {} {:.2f}\n", r.getAccount(),
51           r.getFirst(), r.getLast(), r.getBalance());
52     }
53  }
54
55  int main() {
56     std::vector records{
57        Record{100, "Brian",  "Blue", 123.45},
58        Record{200, "Sue",  "Green", 987.65}
59     };
60
61     std::cout << "Records to serialize:\n";
62     displayRecords(records);
63
64     // serialize vector of Records to JSON and store in text file
65     if (std::ofstream output{"records2.json"}) {
66        cereal::JSONOutputArchive archive{output};
67        archive(cereal::make_nvp("records", records)); // serialize records
68     }
69
70     // deserialize JSON from text file into vector of Records
71     if (std::ifstream input{"records2.json"}) {
72        cereal::JSONInputArchive archive{input};
73        std::vector<Record> deserializedRecords{};
74        archive(deserializedRecords); // deserialize records
75        std::cout << "\nDeserialized records:\n";
76        displayRecords(deserializedRecords);
77     }
78  }
```

Fig. 9.37 Serializing and deserializing objects containing `private` data.

This `Record` class provides a constructor (19–22), *get* member functions (lines 25–28) and `private` data (lines 31–34). There are two key items to note about this class:

- Lines 14–15 declare the function template `serialize` as a `friend` of this class. This enables `serialize` to access directly the `private` data members `account`, `first`, `last` and `balance`.
- The constructor's parameters all have default arguments, allowing `cereal` to use this as the default constructor when deserializing `Record` objects.

The `serialize` function (lines 39–45) now accesses class `Record`'s `private` data members, and the `displayRecords` function (lines 48–53) now uses each `Record`'s *get* functions to access the data to display. The `main` function is identical to the one in Section 9.22.1 and produces the same results, so we do not show the output here.

9.23 Wrap-Up

In this chapter, you created your own classes, created objects of those classes and called member functions of those objects to perform useful actions. You declared data members of a class to maintain data for each object of the class, and you defined member functions to operate on that data. You also learned how to use a class's constructor to specify the initial values for an object's data members.

We used a `Time` class case study to introduce various additional features. We showed how to engineer a class to separate its interface from its implementation. You used the arrow operator to access an object's members via a pointer to an object. You saw that member functions have class scope—the member function's name is known only to the class's other member functions unless referred to by a client of the class via an object name, a reference to an object of the class, a pointer to an object of the class or the scope resolution operator. We also discussed access functions (commonly used to retrieve the values of data members or to test whether a condition is *true* or *false*) and utility functions (`private` member functions that support the operation of the class's `public` member functions).

You saw that a constructor can specify default arguments that enable it to be called multiple ways. You also saw that any constructor that can be called with no arguments is a default constructor. We demonstrated how to share code among constructors with delegating constructors. We discussed destructors for performing termination housekeeping on an object before that object is destroyed, and demonstrated the order in which an object's constructors and destructors are called.

We showed the problems that can occur when a member function returns a reference or a pointer to a `private` data member, which breaks the class's encapsulation. We also showed that objects of the same type can be assigned to one another using the default assignment operator.

You learned how to specify `const` objects and `const` member functions to prevent modifications to objects, thus enforcing the principle of least privilege. You also learned that, through composition, a class can have objects of other classes as members. We demonstrated how to declare and use `friend` functions.

You saw that the `this` pointer is passed as an implicit argument to each of a class's non-`static` member functions, allowing them to access the correct object's data members and other non-`static` member functions. We used the `this` pointer explicitly to access the class's members and to enable cascaded member-function calls. We motivated the notion of `static` data members and member functions and demonstrated how to declare and use them.

We introduced aggregate types and C++20's designated initializers for aggregates. Finally, we presented our next Objects-Natural case study on serializing objects with JSON (JavaScript Object Notation) using the `cereal` library.

In the next chapter, we continue our discussion of classes by introducing inheritance. We'll see classes that share common attributes and behavior can inherit them from a common "base" class. Then, we build on our discussion of inheritance by introducing polymorphism. This object-oriented concept enables us to write programs that handle, in a more general manner, objects of classes related by inheritance.

10. OOP: Inheritance and Runtime Polymorphism

Objectives

In this chapter, you'll:

- Use traditional and modern inheritance idioms, and understand base classes and derived classes.
- Understand the order in which C++ calls constructors and destructors in inheritance hierarchies.
- See how runtime polymorphism can make programming more convenient and systems more easily extensible.
- Use `override` to tell the compiler that a derived-class function overrides a base-class `virtual` function.
- Use `final` at the end of a function's prototype to indicate that function may not be overridden.
- Use `final` after a class's name in its definition to indicate that a class cannot be a base class.
- Perform inheritance with abstract and concrete classes.
- See how C++ can implement `virtual` functions and dynamic binding—and get a sense of `virtual` function overhead.
- Use the non-`virtual` interface idiom (NVI) with `public` non-`virtual` and `private`/`protected` `virtual` functions.
- Use interfaces to create more flexible runtime-polymorphism systems.
- Implement runtime polymorphism without class hierarchies via `std::variant` and `std::visit`.

Outline

10.1 Introduction
10.2 Base Classes and Derived Classes
10.2.1 `CommunityMember` Class Hierarchy
10.2.2 `Shape` Class Hierarchy and `public` Inheritance
10.3 Relationship Between Base and Derived Classes
10.3.1 Creating and Using a `SalariedEmployee` Class
10.3.2 Creating a `SalariedEmployee`–`SalariedCommissionEmployee` Inheritance Hierarchy
10.4 Constructors and Destructors in Derived Classes
10.5 Intro to Runtime Polymorphism: Polymorphic Video Game
10.6 Relationships Among Objects in an Inheritance Hierarchy
10.6.1 Invoking Base-Class Functions from Derived-Class Objects
10.6.2 Aiming Derived-Class Pointers at Base-Class Objects
10.6.3 Derived-Class Member-Function Calls via Base-Class Pointers
10.7 Virtual Functions and Virtual Destructors
10.7.1 Why `virtual` Functions Are Useful
10.7.2 Declaring `virtual` Functions
10.7.3 Invoking a `virtual` Function
10.7.4 `virtual` Functions in the `SalariedEmployee` Hierarchy
10.7.5 `virtual` Destructors
10.7.6 `final` Member Functions and Classes
10.8 Abstract Classes and Pure `virtual` Functions
10.8.1 Pure `virtual` Functions
10.8.2 Device Drivers: Polymorphism in Operating Systems

10.9 Case Study: Payroll System Using Runtime Polymorphism
10.9.1 Creating Abstract Base Class `Employee`
10.9.2 Creating Concrete Derived Class `SalariedEmployee`
10.9.3 Creating Concrete Derived Class `CommissionEmployee`
10.9.4 Demonstrating Runtime Polymorphic Processing
10.10 Runtime Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"
10.11 Non-Virtual Interface (NVI) Idiom
10.12 Program to an Interface, Not an Implementation
10.12.1 Rethinking the Employee Hierarchy: `CompensationModel` Interface
10.12.2 Class `Employee`
10.12.3 `CompensationModel` Implementations
10.12.4 Testing the New Hierarchy
10.12.5 Dependency Injection Design Benefits
10.13 Runtime Polymorphism with `std::variant` and `std::visit`
10.14 Multiple Inheritance
10.14.1 Diamond Inheritance
10.14.2 Eliminating Duplicate Subobjects with `virtual` Base-Class Inheritance
10.15 `protected` Class Members: A Deeper Look
10.16 `public`, `protected` and `private` Inheritance
10.17 More Runtime Polymorphism Techniques; Compile-Time Polymorphism
10.17.1 Other Runtime Polymorphism Techniques
10.17.2 Compile-Time (Static) Polymorphism Techniques
10.17.3 Other Polymorphism Concepts
10.18 Wrap-Up

10.1 Introduction

This chapter continues our object-oriented programming (OOP) discussion by introducing inheritance and runtime polymorphism. With **inheritance**, you'll create classes that absorb existing classes' capabilities, then customize or enhance them.

When creating a class, you can specify that the new class should **inherit** an existing class's members. This existing class is called the **base class**, and the new class is called the **derived class**. Some programming languages, such as Java and C#, use the terms **superclass** and **subclass** for base class and derived class.

Has-a vs. *Is-a* Relationships

We distinguish between the *has-a* relationship and the *is-a* relationship:

- The *has-a* relationship represents composition (Section 9.17) in which an object *contains* as members one or more objects of other classes. For example, a `Car` *has a* steering wheel, *has a* brake pedal, *has an* engine, *has a* transmission, etc.
- The **is-a relationship** represents inheritance. In an *is-a* relationship, a derived-class object also can be treated as an object of its base-class type. For example, a `Car` *is a* `Vehicle`, so a `Car` also exhibits a `Vehicle`'s behaviors and attributes.

Runtime Polymorphism

We'll explain and demonstrate runtime polymorphism with inheritance hierarchies.[1] **Runtime polymorphism enables you to conveniently "program in the general" rather than "program in the specific."** Programs can process objects of classes related by inheritance as if they're all objects of the base-class type. You'll see that runtime polymorphic code—as we'll initially implement it with inheritance and `virtual` functions—refers to objects via base-class pointers or base-class references.

---

1. We'll see later that runtime polymorphism does not have to be done with inheritance hierarchies, and we'll also discuss compile-time polymorphism in Chapter 15, Templates, C++20 Concepts and Metaprogramming.

Implementing for Extensibility

With runtime polymorphism, you can design and implement systems that are more easily **extensible**—new classes can be added with little or no modification to the program's general portions, as long as the new classes are part of the program's inheritance hierarchy. You modify only code that requires direct knowledge of the new classes. Suppose a new class `Car` inherits from class `Vehicle`. We need to write only the new class and code that creates `Car` objects and adds them to the system—the new class simply "plugs right in." The general code that processes `Vehicles` can remain the same.

Discussion of Runtime Polymorphism with Virtual Functions "Under the Hood"

A key feature of this chapter is its discussion of runtime polymorphism, `virtual` functions and dynamic binding "under the hood." The C++ standard does not specify how language features should be implemented. We use a detailed illustration to explain how runtime polymorphism with virtual functions *can* be implemented in C++.

This chapter presents a traditional introduction to inheritance and runtime polymorphism to acquaint you with basic- through intermediate-level thinking and ensure that you understand the mechanics of how these technologies work. Later in this chapter and in the remainder of the book, we'll address current thinking and programming idioms.

Non-Virtual Interface Idiom

Next, we consider another runtime polymorphism approach—the **non-virtual interface idiom (NVI)**, in which each base-class function serves only one purpose:

- as a function that client code can call to perform a task or
- as a function that derived classes can customize.

The customizable functions are internal implementation details of the classes, making systems easier to maintain and evolve without requiring code changes in client applications.

Interfaces and Dependency Injection

To explain the mechanics of inheritance, our first few examples focus on **implementation inheritance**, which is primarily used to define closely related classes with many of the same data members and member-function implementations. For decades, this style of inheritance was widely practiced in the object-oriented programming community. Over time, though, experience revealed its weaknesses. C++ is used to build real-world, business-critical and mission-critical systems—often at a

grand scale. The empirical results from years of building such implementation-inheritance-based systems show that they can be challenging to modify.

So, we'll **refactor** one of our **implementation inheritance** examples to use **interface inheritance. The base class will not provide any implementation details.** Instead, it will contain "placeholders" that tell derived classes the functions they're required to implement. **The derived classes will provide the implementation details**—that is, the data members and member-function implementations. As part of this example, we'll introduce **dependency injection**, in which a class contains a pointer to an object that provides a behavior required by objects of the class—in our example, calculating an employee's earnings. This pointer can be re-aimed—for example, if an employee gets promoted, we can change the earnings calculation by aiming the pointer at an appropriate object. Then, we'll discuss how this refactored example is easier to evolve.

Runtime Polymorphism Via `std::variant` and `std::visit`

17 Next, we'll implement runtime polymorphism for objects of classes that are *not* related by inheritance and do *not* have `virtual` functions. To accomplish this, we'll use C++17's class template `std::variant` and the standard-library function `std::visit`. **Invoking common functionality on objects of unrelated types is called duck typing.**

Multiple Inheritance

We'll demonstrate **multiple inheritance**, which enables a derived class to inherit the members of several base classes. We discuss potential problems with multiple inheritance and how virtual inheritance can solve them.

Other Ways to Implement Polymorphism

20 Finally, we'll overview other runtime-polymorphism techniques and several template-based **compile-time polymorphism** approaches. We'll present compile-time polymorphism in Chapter 15, Templates, C++20 Concepts and Metaprogramming. There you'll see that C++20's new **concepts** feature obviates some older techniques and expands your "toolkit" for creating compile-time, template-based solutions. We'll also provide links to advanced resources for developers who wish to dig deeper.

Chapter Goal

A goal of this chapter is to familiarize you with inheritance and runtime polymorphism mechanics, so you'll be able to better appreciate modern polymorphism idioms and techniques that can promote ease of modifiability and better performance.

10.2 Base Classes and Derived Classes

The following table lists several simple base-class and derived-class examples. Base classes tend to be *more general* and derived classes *more specific*.

Base class	Derived classes
`Student`	`GraduateStudent, UndergraduateStudent`

Base class	Derived classes
`Shape`	`Circle, Triangle, Rectangle, Sphere, Cube`
`Loan`	`CarLoan, HomeImprovementLoan, MortgageLoan`
`Vehicle`	`Car, Motorcycle, Boat`
`Employee`	`Faculty, Staff`
`Account`	`CheckingAccount, SavingsAccount`

Every derived-class object *is an* object of its base-class type, and one base class can have many derived classes. So, the set of objects a base class represents typically is larger than the set of objects a derived class represents. For example, the base class `Vehicle` represents all vehicles, including cars, trucks, boats, airplanes, bicycles, etc. By contrast, the derived-class `Car` represents a smaller, more specific subset of all `Vehicle`s.

10.2.1 `CommunityMember` *Class Hierarchy*

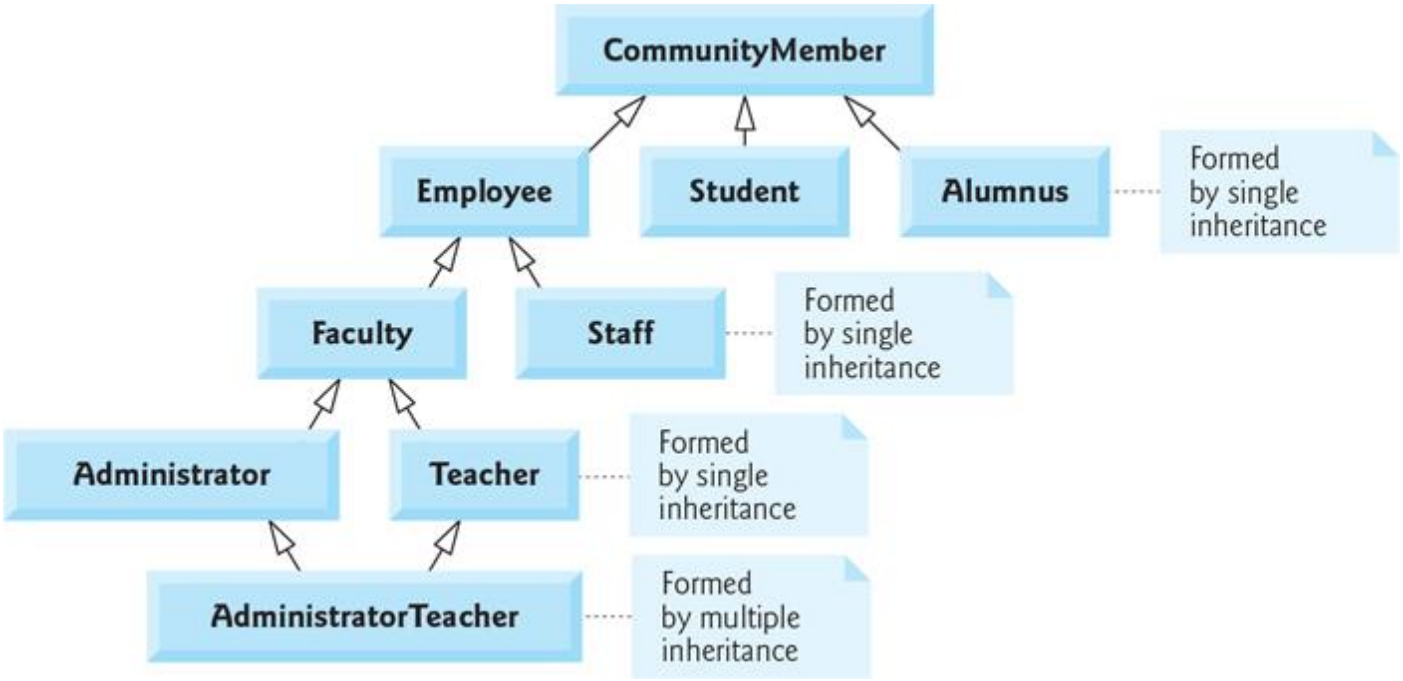
Inheritance relationships naturally form **class hierarchies**. A base class exists in a hierarchical relationship with its derived classes. Once classes are employed in inheritance hierarchies, they become coupled with other classes.[2] A class can be

2. We'll see that tightly coupled classes can make systems difficult to modify. We'll show alternatives to avoid tight coupling.

1. a base class that supplies members to other classes,
2. a derived class that inherits members from other classes or
3. both.[3]

3. Some leaders in the software-engineering community discourage the last option. See Scott Meyers, "Item 33: Make Non-Leaf Classes Abstract," *More Effective C++: 35 New Ways to Improve Your Programs and Designs*. Addison-Wesley, 1995. Also see, Herb Sutter, "Virtuality," *C/C++ Users Journal*, vol. 19, no. 9, September 2001. http://www.gotw.ca/publications/mill18.htm.

Let's develop a simple inheritance hierarchy, represented by the following **UML class diagram**. Such diagrams illustrate the relationships among classes in a hierarchy:



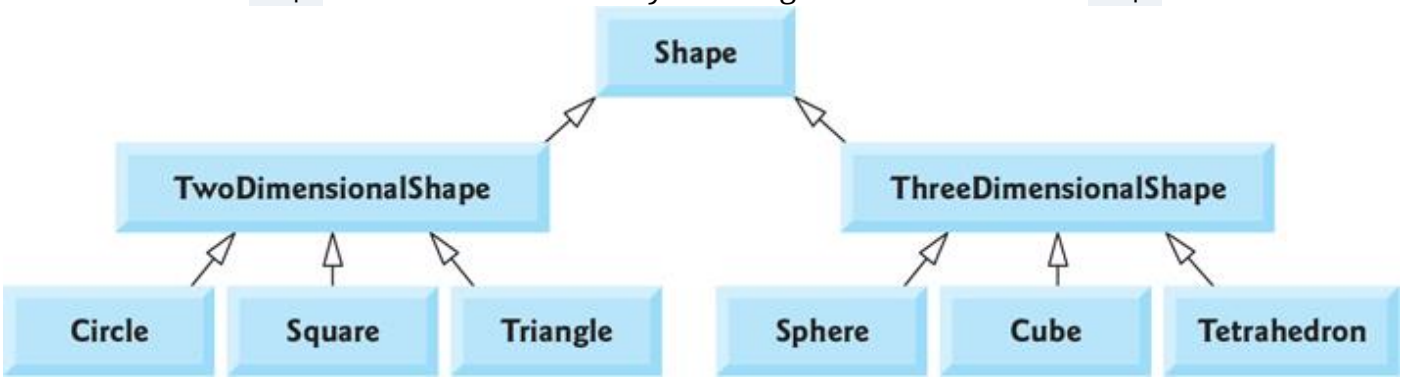
A university community might have thousands of `CommunityMember`s, such as `Employee`s, `Student`s and alumni (each of class `Alumnus`). `Employee`s are either `Faculty` or `Staff`, and `Faculty` are either `Administrator`s or `Teacher`s. Some `Administrator`s are also `Teacher`s. With **single inheritance**, a derived class inherits from *one* base class. With **multiple inheritance**, a derived class inherits from two or more base classes. In this hierarchy, we've used multiple inheritance to form class `AdministratorTeacher`.

Each upward-pointing arrow in the diagram represents an is-a relationship. Following the arrows upward, we can state "an `Employee` *is a* `CommunityMember`" and "a `Teacher` *is a* `Faculty` member." `CommunityMember` is the **direct base class** of `Employee`, `Student` and `Alumnus`. `CommunityMember` is an **indirect base class** of all the hierarchy's other classes. An indirect base class is two or more levels up the class hierarchy from its derived classes.

You can follow the arrows upward several levels, applying the *is-a* relationship. So, an `AdministratorTeacher` *is an* `Administrator` (and also *is a* `Teacher`), *is a* `Faculty` member, *is an* `Employee` and *is a* `CommunityMember`.

10.2.2 `Shape` Class Hierarchy and `public` Inheritance

Let's consider a `Shape` inheritance hierarchy that begins with base-class `Shape`:



Classes `TwoDimensionalShape` and `ThreeDimensionalShape` derive from `Shape`, so a `TwoDimensionalShape` *is a* `Shape`, and a `ThreeDimensionalShape` *is a* `Shape`. The hierarchy's third level contains specific `TwoDimensionalShape`s and `ThreeDimensionalShape`s. We can follow the arrows upward to identify direct and indirect *is-a* relationships. For instance, a `Triangle` *is a* `TwoDimensionalShape` and *is a* `Shape`, while a `Sphere` *is a* `ThreeDimensionalShape` and *is a* `Shape`.

The following class header specifies that `TwoDimensionalShape` inherits from `Shape`:

```
class TwoDimensionalShape : public Shape
```

This is **`public` inheritance**, which we'll use in this chapter. With `public` inheritance, `public` base-class members become `public` derived-class members and `protected` base-class members become `protected` derived-class members (we'll discuss `protected` later). Although `private` base-class members are not directly accessible in derived classes, these members are still inherited and part of the derived-class objects. The derived class can manipulate `private` base-class members through inherited base-class `public` and `protected` member functions—if these base-class member functions provide such functionality. We'll also discuss `private` and `protected` inheritance (Section 10.16).

SE **Inheritance is not appropriate for every class relationship**. Composition's *has-a* relationship sometimes is more appropriate. For example, given `Employee`, `BirthDate` and `PhoneNumber` classes, it's improper to say that an `Employee` *is a* `BirthDate` or that an `Employee` *is a* `PhoneNumber`. However, an `Employee` *has a* `BirthDate` and *has a* `Phone-Number`.

It's possible to treat base-class objects and derived-class objects similarly—their commonalities are expressed in the base-class members. Later in this chapter, we'll consider examples that take advantage of that relationship.

10.3 Relationship Between Base and Derived Classes

This section uses an inheritance hierarchy of employee types in a company's payroll application to demonstrate the relationship between base and derived classes:

- Base-class salaried employees are paid a fixed weekly salary.
- Derived-class salaried commission employees receive a weekly salary *plus* a percentage of their sales.

10.3.1 Creating and Using a `SalariedEmployee` Class

Let's examine `SalariedEmployee`'s class definition (Figs. 10.1–10.2). Its header (Fig. 10.1) specifies the class's `public` services:

- a constructor (line 9),
- member function `earnings` (line 17),
- member function `toString` (line 18) and
- `public` *get* and *set* functions that manipulate the class's data members `m_name` and `m_salary` (declared in lines 20–21).

```
1  // Fig. 10.1: SalariedEmployee.h
2  // SalariedEmployee class definition.
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5  #include <string_view>
6
7  class SalariedEmployee {
8  public:
9     SalariedEmployee(std::string_view name, double salary);
```

```
10
11      void setName(std::string_view name);
12      std::string getName() const;
13
14      void setSalary(double salary);
15      double getSalary() const;
16
17      double earnings() const;
18      std::string toString() const;
19   private:
20      std::string m_name{};
21      double m_salary{0.0};
22   };
```

Fig. 10.1 `SalariedEmployee` class definition.

Member function `setSalary`'s implementation (Fig. 10.2, lines 22–28) validates its argument before modifying the data member `m_salary`.

```
1   // Fig. 10.2: SalariedEmployee.cpp
2   // Class SalariedEmployee member-function definitions.
3   #include <fmt/format.h>
4   #include <stdexcept>
5   #include "SalariedEmployee.h" // SalariedEmployee class definition
6
7   // constructor
8   SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
9      : m_name{name} {
10     setSalary(salary);
11  }
12
13  // set name
14  void SalariedEmployee::setName(std::string_view name) {
15     m_name = name; // should validate
16  }
17
18  // return name
19  std::string SalariedEmployee::getName() const {return m_name;}
20
21  // set salary
22  void SalariedEmployee::setSalary(double salary) {
23     if (salary < 0.0) {
24        throw std::invalid_argument("Salary must be >= 0.0");
25     }
26
27     m_salary = salary;
28  }
29
30  // return salary
31  double SalariedEmployee::getSalary() const {return m_salary;}
32
33  // calculate earnings
34  double SalariedEmployee::earnings() const {return getSalary();}
35
36  // return string representation of SalariedEmployee object
37  std::string SalariedEmployee::toString() const {
```

```
38      return fmt::format("name: {}\nsalary: ${:.2f}\n", getName(),
39         getSalary());
40   }
```

Fig. 10.2 Class `SalariedEmployee` member-function definitions.

`SalariedEmployee` Constructor

The class's constructor (Fig. 10.2, lines 8–11) uses a member-initializer list to initialize `m_name`. We could validate the name, perhaps by ensuring that it's of a reasonable length. The constructor calls `setSalary` to validate and initialize data member `m_salary`.

`SalariedEmployee` Member Functions `earnings` and `toString`

Function `earnings` (line 34) calls `getSalary` and returns the result. Function `toString` (lines 37–40) returns a `string` containing a `SalariedEmployee` object's information.

Testing Class `SalariedEmployee`

Figure 10.3 tests class `SalariedEmployee`. Line 9 creates the `SalariedEmployee` object `employee`. Lines 12–14 demonstrate the `employee`'s *get* functions. Line 16 uses `setSalary` to change the `employee`'s `m_salary` value. Then, lines 17–18 call `employee`'s `toString` member function to get and output the `employee`'s updated information. Finally, line 21 displays the `employee`'s `earnings`, using the updated `m_salary` value.

```
1   // fig10_03.cpp
2   // SalariedEmployee class test program.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include "SalariedEmployee.h" // SalariedEmployee class definition
6
7   int main() {
8      // instantiate a SalariedEmployee object
9      SalariedEmployee employee{"Sue Jones", 300.0};
10
11     // get SalariedEmployee data
12     std::cout << "Employee information obtained by get functions:\n"
13          << fmt::format("name: {}\nsalary: ${:.2f}\n", employee.getName(),
14               employee.getSalary());
15
16     employee.setSalary(500.0); // change salary
17     std::cout << "\nUpdated employee information from function toString:\n"
18        << employee.toString();
19
20     // display only the employee's earnings
21     std::cout << fmt::format("\nearnings: ${:.2f}\n", employee.earnings());
22  }
```

```
Employee information obtained by get functions:
name: Sue Jones
salary: $300.00

Updated employee information from function toString:
name: Sue Jones
salary: $500.00

earnings: $500.00
```

Fig. 10.3 `SalariedEmployee` class test program.

10.3.2 Creating a `SalariedEmployee`*–*`SalariedCommissionEmployee` *Inheritance Hierarchy*

Now, let's create a `SalariedCommissionEmployee` class (Figs. 10.4–10.5) that inherits from `SalariedEmployee` (Figs. 10.1–10.2). In this example, a `SalariedCommissionEmployee` object *is a* `SalariedEmployee`—`public` inheritance passes on `SalariedEmployee`'s capabilities. A `SalariedCommissionEmployee` also has `m_grossSales` and `m_commission-Rate` data members (Fig. 10.4, lines 22–23), which we'll multiply to calculate the commission earned. Lines 13–17 declare `public` *get* and *set* functions that manipulate the class's `m_grossSales` and `m_commissionRate` data members.

```
1  // Fig. 10.4: SalariedCommissionEmployee.h
2  // SalariedCommissionEmployee class derived from class SalariedEmployee.
3  #pragma once
4  #include <string>
5  #include <string_view>
6  #include "SalariedEmployee.h"
7
8  class SalariedCommissionEmployee : public SalariedEmployee {
9  public:
10    SalariedCommissionEmployee(std::string_view name, double salary,
11       double grossSales, double commissionRate);
12
13    void setGrossSales(double grossSales);
14    double getGrossSales() const;
15
16    void setCommissionRate(double commissionRate);
17    double getCommissionRate() const;
18
19    double earnings() const;
20    std::string toString() const;
21 private:
22    double m_grossSales{0.0};
23    double m_commissionRate{0.0};
24 };
```

Fig. 10.4 `SalariedCommissionEmployee` class definition indicating inheritance relationship with class `CommissionEmployee`.

Inheritance

SE The **colon (:)** in line 8 indicates inheritance, and `public` to its right specifies the kind of inheritance. In `public` inheritance, `public` base-class members remain `public` in the derived class and `protected` ones (discussed later) remain `protected`. `SalariedCommissionEmployee` inherits all of `SalariedEmployee`'s members, *except* its constructor(s) and destructor. **Constructors and destructors are specific to the class that defines them, so derived classes have their own.**[4]

4. It's common in a derived-class constructor to simply pass its arguments to a corresponding base-class constructor and do nothing else. Online Chapter 20, Other Topics, shows how to inherit a base class's constructors.

`SalariedCommissionEmployee` Member Functions

Class `SalariedCommissionEmployee`'s `public` services (Fig. 10.4) include

- its own constructor (lines 10–11),
- the member functions `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` and `toString` (lines 13–20), and
- the `public` member functions inherited from class `SalariedEmployee`.

Although `SalariedCommissionEmployee`'s source code does not contain these inherited members, they're nevertheless a part of the class. A `SalariedCommissionEmployee` object also contains `SalariedEmployee`'s `private` members, but **they're not directly accessible** within the derived class. You can access them only via the inherited `SalariedEmployee public` (or `protected`) member functions.

`SalariedCommissionEmployee`'s Implementation

Figure 10.5 shows `SalariedCommissionEmployee`'s member-function implementations. **Each derived-class constructor must call a base-class constructor.** We do this explicitly via a **base-class initializer** (line 11)—a member initializer that passes arguments to a base-class constructor. In the `SalariedCommissionEmployee` constructor (lines 8–15), line 11 calls `SalariedEmployee`'s constructor to initialize the inherited data members with the arguments `name` and `salary`. A base-class initializer invokes the base-class constructor by name. Functions `setGrossSales` (lines 18–24) and `setCommissionRate` (lines 32–41) validate their arguments before modifying the `m_grossSales` and `m_commissionRate` data members.

```
1  // Fig. 10.5: SalariedCommissionEmployee.cpp
2  // Class SalariedCommissionEmployee member-function definitions.
3  #include <fmt/format.h>
4  #include <stdexcept>
5  #include "SalariedCommissionEmployee.h"
6
7  // constructor
8  SalariedCommissionEmployee::SalariedCommissionEmployee(
9     std::string_view name, double salary, double grossSales,
10    double commissionRate)
11    : SalariedEmployee{name, salary} { // call base-class constructor
12
13    setGrossSales(grossSales); // validate & store gross sales
14    setCommissionRate(commissionRate); // validate & store commission rate
15 }
16
17 // set gross sales amount
18 void SalariedCommissionEmployee::setGrossSales(double grossSales) {
19    if (grossSales < 0.0) {
20       throw std::invalid_argument("Gross sales must be >= 0.0");
21    }
22
23    m_grossSales = grossSales;
24 }
25
26 // return gross sales amount
```

```
27   double SalariedCommissionEmployee::getGrossSales() const {
28      return m_grossSales;
29   }
30
31   // return commission rate
32   void SalariedCommissionEmployee::setCommissionRate(
33      double commissionRate) {
34
35      if (commissionRate <= 0.0 || commissionRate >= 1.0) {
36         throw std::invalid_argument(
37            "Commission rate must be > 0.0 and < 1.0");
38      }
39
40      m_commissionRate = commissionRate;
41   }
42
43   // get commission rate
44   double SalariedCommissionEmployee::getCommissionRate() const {
45      return m_commissionRate;
46   }
47
48   // calculate earnings--uses SalariedEmployee::earnings()
49   double SalariedCommissionEmployee::earnings() const {
50      return SalariedEmployee::earnings() +
51         getGrossSales() * getCommissionRate();
52   }
53
54   // returns string representation of SalariedCommissionEmployee object
55   std::string SalariedCommissionEmployee::toString() const {
56      return fmt::format(
57         "{}gross sales: ${:.2f}\ncommission rate: {:.2f}\n",
58         SalariedEmployee::toString(), getGrossSales(), getCommissionRate());
59   }
```

Fig. 10.5 Class `SalariedCommissionEmployee` member-function definitions.

`SalariedCommissionEmployee` Member Function `earnings`

`SalariedCommissionEmployee`'s `earnings` function (lines 49–52) redefines `earnings` from class `SalariedEmployee` (Fig. 10.2, line 34) to calculate a `SalariedCommissionEmployee`'s earnings. Line 50 in `SalariedCommissionEmployee`'s version uses the expression

```
SalariedEmployee::earnings()
```

to get the portion of the earnings based on salary, then adds that value to the commission to calculate the total earnings.

SE Err **To call a redefined base-class member function from the derived class, place the base-class name and the scope resolution operator (`::`) before the base-class member-function name. `SalariedEmployee::` is required here to avoid infinite recursion.** Also, **we avoid duplicating code** by calling `SalariedEmployee`'s `earnings` function from `SalariedCommissionEmployee`'s `earnings` function.[5]

5. Avoiding code duplication is the upside. The downside is that this creates a coupling between base and derived classes that can make large-scale systems difficult to modify.

`SalariedCommissionEmployee` Member Function `toString`

Similarly, `SalariedCommissionEmployee`'s `toString` function (Fig. 10.5, lines 55–59) **redefines** class `SalariedEmployee`'s `toString` function (Fig. 10.2, lines 37–40). The new version returns a `string` containing

- the result of calling `SalariedEmployee::toString()` (Fig. 10.5, line 58) and
- the `SalariedCommissionEmployee`'s gross sales and commission rate.

Testing Class `SalariedCommissionEmployee`

Figure 10.6 creates the `SalariedCommissionEmployee` object `employee` (line 9). Lines 12–16 output the `employee`'s data by calling its *get* functions to retrieve the object's data member values. Lines 18–19 use `setGrossSales` and `setCommissionRate` to change the `employee`'s `m_grossSales` and `m_commissionRate` values, respectively. Then, lines 20–21 call `employee`'s `toString` member function to show the `employee`'s updated information. Finally, line 24 displays the `employee`'s updated earnings.

```
1  // fig10_06.cpp
2  // SalariedCommissionEmployee class test program.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include "SalariedCommissionEmployee.h"
6
7  int main() {
8     // instantiate SalariedCommissionEmployee object
9     SalariedCommissionEmployee employee{"Bob Lewis", 300.0, 5000.0, .04};
10
11    // get SalariedCommissionEmployee data
12    std::cout << "Employee information obtained by get functions:\n"
13         << fmt::format("{}: {}\n{}: {:.2f}\n{}: {:.2f}\n{}: {:.2f}\n",
14           "name", employee.getName(), "salary", employee.getSalary(),
15           "gross sales", employee.getGrossSales(),
16           "commission", employee.getCommissionRate());
17
18    employee.setGrossSales(8000.0); // change gross sales
19    employee.setCommissionRate(0.1); // change commission rate
20    std::cout << "\nUpdated employee information from function toString:\n"
21       << employee.toString();
22
23    // display the employee's earnings
24    std::cout << fmt::format("\nearnings: ${:.2f}\n", employee.earnings());
25 }
```

```
Employee information obtained by get functions:
name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission: 0.04

Updated employee information from function toString:
name: Bob Lewis
```

```
salary: $300.00
gross sales: $8000.00
commission rate: 0.10

earnings: $1100.00
```

Fig. 10.6 `SalariedCommissionEmployee` class test program.

A Derived-Class Constructor Must Call Its Base Class's Constructor

Err The compiler would issue an error if `SalariedCommissionEmployee`'s constructor did not explicitly invoke class `SalariedEmployee`'s constructor. In this case, the compiler would attempt to call class `SalariedEmployee`'s default constructor, which does not exist because the base class explicitly defined a constructor. **If the base class provides a default constructor, the derived-class constructor can call the base-class constructor implicitly.**

Notes on Constructors in Derived Classes

SE Err *A derived-class constructor must call its base class's constructor with any required arguments*; otherwise, a compilation error occurs. **This ensures that inherited `private` base-class members, which the derived class cannot access directly, get initialized.** The derived class's data-member initializers are typically placed after the base-class initializer(s) in the member-initializer list.

Base-Class `private` Members Are Not Directly Accessible in a Derived Class

Err C++ rigidly enforces restrictions on accessing `private` data members. **Even a derived class, which is intimately related to its base class, cannot directly access its base class's `private` data.** For example, class `SalariedEmployee`'s `private m_salary` data member, though part of each `SalariedCommissionEmployee` object, cannot be accessed directly by class `SalariedCommissionEmployee`'s member functions. If they try, the compiler produces an error message, such as the following from GNU `g++`:

```
‘double SalariedEmployee::m_salary’ is private within this context
```

However, as you saw in Fig. 10.5, `SalariedCommissionEmployee`'s member functions can access the `public` members inherited from class `SalariedEmployee`.

Including the Base-Class Header in the Derived-Class Header

Notice that we `#include` the base class's header in the derived class's header (Fig. 10.4, line 6). This is necessary for several reasons:

- For the derived class to inherit from the base class in line 8 (Fig. 10.4), the compiler requires the base-class definition from `SalariedEmployee.h`.
- SE **The compiler determines an object's size from its class's definition.** To create an object, the compiler must know the class definition to reserve the proper amount of memory. **A derived-class object's size depends on the data members declared explicitly in its class definition and the data members inherited from its direct and indirect base classes.**[6] Including the base class's definition

allows the compiler to determine the complete memory requirements for all the data members that contribute to a derived-class object's total size.

---

6. All objects of a class share one copy of the class's member functions, which are stored separately from those objects and are not part of their size.

- The base-class definition also enables the compiler to determine whether the derived class uses the base class's inherited members properly. For example, **the compiler uses the base class's function prototypes to validate derived-class function calls to inherited base-class functions**.

Eliminating Repeated Code via Implementation Inheritance

SE With implementation inheritance, the base class declares the common data members and member functions of all the classes in the hierarchy. When changes are required for these common features, you make the changes only in the base class. The derived classes then inherit the changes and must be recompiled. Without inheritance, changes would need to be made to all the source-code files that contain a copy of the code in question.[7]

---

7. Again, the downside of implementation inheritance, especially in deep inheritance hierarchies, is that it creates a tight coupling among the classes in the inheritance hierarchy, making it difficult to modify.

10.4 Constructors and Destructors in Derived Classes

Order of Constructor Calls

SE Instantiating a derived-class object begins a **chain of constructor calls**. **Before performing its own tasks, a derived-class constructor invokes its direct base class's constructor either explicitly via a base-class member initializer or implicitly by calling the base class's default constructor**. The last constructor called in this chain is for the base class at the top of the hierarchy. That constructor finishes executing *first*, and the most-derived-class constructor's body finishes executing *last*.

Each base-class constructor initializes the base-class data members that its derived classes inherit. In the `SalariedEmployee`/`SalariedCommissionEmployee` hierarchy we've been studying, when we create a `SalariedCommissionEmployee` object:

- Its constructor immediately calls the `SalariedEmployee` constructor.
- `SalariedEmployee` is this hierarchy's base class, so the `SalariedEmployee` constructor executes, initializing the `SalariedCommissionEmployee` object's inherited `SalariedEmployee` `m_name` and `m_salary` data members.
- `SalariedEmployee`'s constructor then returns control to `SalariedCommission-Employee`'s constructor to initialize `m_grossSales` and `m_commissionRate`.

Order of Destructor Calls

SE When a derived-class object is destroyed, the program calls that object's destructor. This begins a **chain of destructor calls**. **The destructors execute in the reverse order of the constructors.** When a derived-class object's destructor is called, it performs its task, then invokes the destructor of the next class up the hierarchy. This repeats until the destructor of the base class at the hierarchy's top is called.

Constructors and Destructors for Composed Objects

SE Suppose we create a derived-class object where both the base class and the derived class are composed of objects of other classes. When a derived-class object is created

- the constructors for the base class's member objects execute in the order those objects were declared,
- then the base-class constructor body executes,
- then the constructors for the derived class's member objects execute in the order those objects were declared in the derived class,
- then the derived class's constructor body executes.

Destructors for data members are called in the reverse order that their corresponding constructors were called.

10.5 Intro to Runtime Polymorphism: Polymorphic Video Game

Suppose we're designing a video game containing `Martian`s, `Venusian`s, `Plutonian`s, `SpaceShip`s and `LaserBeam`s. Each inherits from a `SpaceObject` base class with the member function `draw` and implements this function in a manner appropriate to the derived class.

Screen Manager

A screen-manager program maintains a `vector` of `SpaceObject` pointers to objects of the various classes. **To refresh the screen, the screen manager periodically sends each object the same `draw` message. Each object responds in its unique way.** For example,

- a `Martian` might draw itself in red with the appropriate number of antennae,
- a `SpaceShip` might draw itself as a silver flying saucer and
- a `LaserBeam` might draw itself as a bright red beam across the screen.

The same `draw` message has **many forms** of results—hence the term **polymorphism**.

Adding New Classes to the System

A polymorphic screen manager facilitates adding new classes to a system with minimal code modifications. Suppose we want to add `Mercurian` objects to our game. We build a class `Mercurian` that inherits from `SpaceObject` and defines `draw`, then add the object's address to the `vector` of `SpaceObject` pointers. The screen-manager code invokes member function `draw` the same way for every object the `vector` points to, regardless of the object's type. So, the new `Mercurian` objects just "**plug right in.**" **Without modifying the system—other than to create objects of the new classes—you can use runtime polymorphism to accommodate additional classes, including ones not envisioned when the system was created.**

SE Runtime polymorphism enables you to deal in **generalities** and let the execution-time environment concern itself with the **specifics**. You can direct objects to behave in appropriate manners without knowing their types, as long as they belong to the same class hierarchy and are accessed via a common base-class pointer or reference.[8]

---

8. We'll see that there are forms of runtime polymorphism and compile-time polymorphism that do not depend on inheritance hierarchies.

SE Runtime polymorphism promotes **extensibility**. Software that invokes polymorphic behavior is written independently of the specific object types to which messages are sent. Thus, new object types that can respond to existing messages can simply be plugged into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

10.6 Relationships Among Objects in an Inheritance Hierarchy

Section 10.3 created a `SalariedEmployee`–`SalariedCommissionEmployee` class hierarchy. Let's examine the relationships among classes in an inheritance hierarchy more closely. The next several sections present examples demonstrating how base-class and derived-class pointers can be aimed at base-class and derived-class objects, and how those pointers can be used to invoke member functions that manipulate those objects:

- In Section 10.6.1, we assign a derived-class object's address to a base-class pointer, then show that invoking a function via the base-class pointer invokes the *base-class* functionality in the derived-class object. **The handle's type determines which function is called.**
- Err In Section 10.6.2, we assign a base-class object's address to a derived-class pointer to show the resulting compilation error. We discuss the error message and investigate why the compiler does not allow such an assignment.
- Err In Section 10.6.3, we assign a derived-class object's address to a base-class pointer, then examine how the base-class pointer can be used to invoke only the base-class functionality. **Compilation errors occur when we attempt to invoke derived-class-only member functions through the base-class pointer.**
- Finally, Section 10.7 introduces **virtual functions** to demonstrate how to get **runtime polymorphic behavior** from base-class pointers aimed at derived-class objects. We then **assign a derived-class object's address to a base-class pointer and use that pointer to invoke derived-class functionality—precisely what we need to achieve runtime polymorphic behavior.**

These examples demonstrate that **with `public` inheritance, an object of a derived class can be treated as an object of its base class.** This enables various interesting manipulations. For example, a program can create a `vector` of base-class pointers that point to objects of many derived-class types. The compiler allows this because each derived-class object *is an* object of its base class.

SE On the other hand, **you cannot treat a base-class object as an object of a derived class.** For example, a `SalariedEmployee` is not a `SalariedCommissionEmployee`—it's missing the data members `m_grossSales` or `m_commissionRate` and does not have their corresponding *set* and *get* member

functions. **The is-a relationship applies only from a derived class to its direct and indirect base classes.**

10.6.1 Invoking Base-Class Functions from Derived-Class Objects

Figure 10.7 reuses classes `SalariedEmployee` and `SalariedCommissionEmployee` from Sections 10.3.1–10.3.2. The example demonstrates aiming base-class and derived-class pointers at base-class and derived-class objects. The first two are natural and straightforward:

- We aim a base-class pointer at a base-class object and invoke base-class functionality.
- We aim a derived-class pointer at a derived-class object and invoke derived-class functionality.

```
1  // fig10_07.cpp
2  // Aiming base-class and derived-class pointers at base-class
3  // and derived-class objects, respectively.
4  #include <fmt/format.h>
5  #include <iostream>
6  #include "SalariedEmployee.h"
7  #include "SalariedCommissionEmployee.h"
8
9  int main() {
10    // create base-class object
11    SalariedEmployee salaried{"Sue Jones", 500.0};
12
13    // create derived-class object
14    SalariedCommissionEmployee salariedCommission{
15      "Bob Lewis", 300.0, 5000.0, .04};
16
17    // output objects salaried and salariedCommission
18    std::cout << fmt::format("{}:\n{}\n{}\n",
19      "DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS",
20      salaried.toString(), // base-class toString
21      salariedCommission.toString()); // derived-class toString
22
23   // natural: aim base-class pointer at base-class object
24   SalariedEmployee* salariedPtr{&salaried};
25   std::cout << fmt::format("{}\n{}:\n{}\n",
26     "CALLING TOSTRING WITH BASE-CLASS POINTER TO",
27     "BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
28     salariedPtr->toString()); // base-class version
29
30   // natural: aim derived-class pointer at derived-class object
31   SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};
32
33   std::cout << fmt::format("{}\n{}:\n{}\n",
34     "CALLING TOSTRING WITH DERIVED-CLASS POINTER TO",
35     "DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
36     salariedCommissionPtr->toString()); // derived-class version
37
38   // aim base-class pointer at derived-class object
39   salariedPtr = &salariedCommission;
40   std::cout << fmt::format("{}\n{}:\n{}\n",
41     "CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS",
42     "OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
```

```
43         salariedPtr->toString()); // baseclass version
44    }
```

```
DISPLAY BASE-CLASS AND DERIVED-CLASS OBJECTS:
name: Sue Jones
salary: $500.00

name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04

CALLING TOSTRING WITH BASE-CLASS POINTER TO
BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY:
name: Sue Jones
salary: $500.00

CALLING TOSTRING WITH DERIVED-CLASS POINTER TO
DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04

CALLING TOSTRING WITH BASE-CLASS POINTER TO DERIVED-CLASS
OBJECT INVOKES BASE-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
```

Fig. 10.7 Assigning addresses of base-class and derived-class objects to base-class and derived-class pointers.

Then, we demonstrate the *is-a* relationship between derived classes and base classes by aiming a base-class pointer at a derived-class object and showing that **the base-class functionality is available in the derived-class object**.

Recall that a `SalariedCommissionEmployee` object *is a* `SalariedEmployee` who earns a commission based on gross sales. `SalariedCommissionEmployee`'s `earnings` member function (Fig. 10.5, lines 49–52) **redefines** `SalariedEmployee`'s version (Fig. 10.2, line 34) to include the commission. `SalariedCommissionEmployee`'s `toString` member function (Fig. 10.5, lines 55–59) **redefines** `SalariedEmployee`'s version (Fig. 10.2, lines 37–40) to return the same information and the employee's commission.

Creating Objects and Displaying Their Contents

In Fig. 10.7, line 11 creates a `SalariedEmployee` object and lines 14–15 create a `SalariedCommissionEmployee` object. Lines 20 and 21 use each object's name to invoke its `toString` member function.

Aiming a Base-Class Pointer at a Base-Class Object

Line 24 initializes the `SalariedEmployee` pointer `salariedPtr` with the base-class object `salaried`'s address. Line 28 uses the pointer to invoke the `salaried` object's `toString` member function from base-class `SalariedEmployee`.

Aiming a Derived-Class Pointer at a Derived-Class Object

Line 31 initializes `SalariedCommissionEmployee` pointer `salariedCommissionPtr` with derived-class object `salariedCommission`'s address. Line 36 uses the pointer to invoke the `salariedCommission` object's `toString` member function from derived-class `Salaried-CommissionEmployee`.

Aiming a Base-Class Pointer at a Derived-Class Object

Line 39 assigns derived-class object `salariedCommission`'s address to `salariedPtr`—a base-class pointer. **This "crossover" is allowed because a derived-class object is an object of its base class.** Line 43 uses this pointer to invoke member function `toString`. Even though the base-class `SalariedEmployee` pointer points to a `SalariedCommissionEmployee` derived-class object, **the base class's `toString` member function is invoked**. The gross sales and the commission rate are not displayed because they are not base-class members.

This program's output shows that **the invoked function depends on the pointer type (or reference type, as you'll see) used to invoke the function**, *not* the object type for which the member function is called. In Section 10.7, we'll see that **`virtual` functions make it possible to invoke the object type's functionality**—an important aspect of implementing runtime polymorphic behavior.

10.6.2 Aiming Derived-Class Pointers at Base-Class Objects

Now, let's try to aim a derived-class pointer at a base-class object (Fig. 10.8). Line 7 creates a `SalariedEmployee` object. Line 11 attempts to initialize a `SalariedCommissionEm-ployee` pointer with the base-class `salaried` object's address. The compiler generates an error because a `SalariedEmployee` is *not* a `SalariedCommissionEmployee`.

```
1   // fig10_08.cpp
2   // Aiming a derived-class pointer at a base-class object.
3   #include "SalariedEmployee.h"
4   #include "SalariedCommissionEmployee.h"
5
6   int main() {
7      SalariedEmployee salaried{"Sue Jones", 500.0};
8
9      // aim derived-class pointer at base-class object
10     // Error: a SalariedEmployee is not a SalariedCommissionEmployee
11     SalariedCommissionEmployee* salariedCommissionPtr{&salaried};
12  }
```

Microsoft Visual C++ compiler error message:

```
fig10_08.cpp(11,63): error C2440: 'initializing': cannot convert from
'SalariedEmployee *' to 'SalariedCommissionEmployee *'
```

Fig. 10.8 Aiming a derived-class pointer at a base-class object.

Err Consider the consequences if the compiler were to allow this assignment. Through a `SalariedCommissionEmployee` pointer, we can invoke any of that class's member functions, including `setGrossSales` and `setCommissionRate`, for the object to which the pointer points—the base-class object `salaried`. However, a `SalariedEmployee` object has neither `setGrossSales` and `setCommissionRate` member functions nor data members `m_grossSales` and `m_commissionRate` to set. If this were allowed, it could lead to problems

because member functions `setGrossSales` and `setCommissionRate` would assume data members `m_grossSales` and `m_commissionRate` exist at their "usual locations" in a `SalariedCommissionEmployee` object. **A `SalariedEmployee` object's memory does not have these data members, so `setGrossSales` and `setCommissionRate` might overwrite other data in memory.**

10.6.3 Derived-Class Member-Function Calls via Base-Class Pointers

Err **The compiler allows us to invoke only base-class member functions via a base-class pointer.** So, if a base-class pointer is aimed at a derived-class object, and an attempt is made to access a derived-class-only member function, a compilation error occurs. Figure 10.9 shows the compiler errors for invoking a derived-class-only member function via a base-class pointer (lines 22–24).

```
1   // fig10_09.cpp
2   // Attempting to call derived-class-only functions
3   // via a base-class pointer.
4   #include <string>
5   #include "SalariedEmployee.h"
6   #include "SalariedCommissionEmployee.h"
7
8   int main() {
9      SalariedCommissionEmployee salariedCommission{
10        "Bob Lewis", 300.0, 5000.0, .04};
11
12     // aim base-class pointer at derived-class object (allowed)
13     SalariedEmployee* salariedPtr{&salariedCommission};
14
15     // invoke base-class member functions on derived-class
16     // object through base-class pointer (allowed)
17     std::string name{salariedPtr->getName()};
18     double salary{salariedPtr->getSalary()};
19
20     // attempt to invoke derived-class-only member functions
21     // on derived-class object through base-class pointer (disallowed)
22     double grossSales{salariedPtr->getGrossSales()};
23     double commissionRate{salariedPtr->getCommissionRate()};
24     salariedPtr->setGrossSales(8000.0);
25  }
```

GNU C++ compiler error messages:

```
fig10_09.cpp: In function 'int main()':
fig10_09.cpp:22:35: error: 'class SalariedEmployee' has no member named
'getGrossSales'
   22 | double grossSales{salariedPtr->getGrossSales()};
      |                                ^~~~~~~~~~~~~
fig10_09.cpp:23:39: error: 'class SalariedEmployee' has no member named
'getCommissionRate'
   23 | double commissionRate{salariedPtr->getCommissionRate()};
      |                                       ^~~~~~~~~~~~~~~~~
fig10_09.cpp:24:17: error: 'class SalariedEmployee' has no member named
'setGrossSales'
```

```
   24 |     salariedPtr->setGrossSales(8000.0);
      |                  ^~~~~~~~~~~~~
```

Fig. 10.9 Attempting to call derived-class-only functions via a base-class pointer.

Lines 9–10 create a `SalariedCommissionEmployee` object. Line 13 initializes the base-class pointer `salariedPtr` with the derived-class object `salariedCommission`'s address. Again, this is allowed because a `SalariedCommissionEmployee` *is a* `Salaried-Employee`.

Lines 17–18 invoke base-class member functions via the base-class pointer. These calls are allowed because `SalariedCommissionEmployee` inherits these functions.

We know that `salariedPtr` is aimed at a `SalariedCommissionEmployee` object, so lines 22–24 try to invoke `SalariedCommissionEmployee`-only member functions `getGrossSales`, `getCommissionRate` and `setGrossSales`. The compiler generates errors because these functions are not base-class `SalariedEmployee` member functions. Through `salariedPtr`, we can invoke only member functions of the handle's class type—in this case, those of base-class `SalariedEmployee`.

10.7 Virtual Functions and Virtual Destructors

In Section 10.6.1, we aimed a base-class `SalariedEmployee` pointer at a derived-class `SalariedCommissionEmployee` object, then used it to invoke member function `toString`. In that case, the `SalariedEmployee` class's `toString` was called. How can we invoke the derived-class `toString` function via a base-class pointer?

10.7.1 Why `virtual` Functions Are Useful

SE Suppose the shape classes `Circle`, `Triangle`, `Rectangle` and `Square` derive from base-class `Shape`. Each class might be endowed with the ability to draw objects of that class via a member function `draw`, but each shape's implementation is quite different. In a program that draws many different shapes, it would be convenient to treat them all generally as base-class `Shape` objects. We could then draw any shape by using a base-class `Shape` pointer to invoke function `draw`. At runtime, the program would dynamically determine which derived-class `draw` function to use, based on the type of the object to which the base-class `Shape` pointer points. This is runtime polymorphic behavior. **With `virtual` functions, the type of the object pointed to (or referenced)—not the type of the pointer (or reference)—determines which member function to invoke.**

10.7.2 Declaring `virtual` Functions

SE To enable this runtime polymorphic behavior, we declare the base-class member function `draw` as `virtual`,[9] then **override** it in each derived class to draw the appropriate shape. **An overridden function in a derived class must have the same signature as the base-class function it overrides.** To declare a `virtual` function, precede its prototype with the keyword `virtual`. For example,

```
virtual void draw() const;
```

9. Some programming languages, such as Java and Python, treat all member functions (methods) like C++ `virtual` functions. As we'll see in Section 10.10, `virtual` functions have a slight execution-time

performance hit and a slight memory consumption hit. C++ allows you to choose whether to make each function `virtual`, based on the performance requirements of your applications.

SE would appear in base class `Shape`. The preceding prototype declares that function `draw` is a `virtual` function that takes no arguments and returns nothing. This function is declared `const` because a `draw` function should not modify the `Shape` object on which it's invoked. Virtual functions do not have to be `const` and can receive arguments and return values as appropriate. **Once a function is declared `virtual`, it's `virtual` in all classes derived directly or indirectly from that base class.** If a derived class does not override a `virtual` function from its base class, the derived class simply inherits its base class's `virtual` function implementation.

10.7.3 Invoking a `virtual` Function

If a program invokes a `virtual` function through

- a base-class pointer to a derived-class object (e.g., `shapePtr->draw()`) or
- a base-class reference to a derived-class object (e.g., `shapeRef.draw()`),

the program will choose the correct *derived-class* function at execution time, based on the object's type—*not* the pointer or reference type. Choosing the appropriate function to call at execution time is known as **dynamic binding** or **late binding**.

When a `virtual` function is called via a specific object's *name* using the dot operator (e.g., `squareObject.draw()`), an optimizing compiler can resolve at *compile-time* the function to call. This is called **static binding**. The `virtual` function that will be called is the one defined for that object's class.

10.7.4 `virtual` Functions in the `SalariedEmployee` Hierarchy

Let's see how `virtual` functions could enable runtime polymorphic behavior in our hierarchy. We made only two modifications to each class's header to enable the behavior you'll see in Fig. 10.10. In class `SalariedEmployee`'s header (Fig. 10.1), we modified the `earnings` and `toString` prototypes (lines 17–18)

```
double earnings() const;
std::string toString() const;
```

```
 1  // fig10_10.cpp
 2  // Introducing polymorphism, virtual functions and dynamic binding.
 3  #include <fmt/format.h>
 4  #include <iostream>
 5  #include "SalariedEmployee.h"
 6  #include "SalariedCommissionEmployee.h"
 7
 8  int main() {
 9     // create base-class object
10     SalariedEmployee salaried{"Sue Jones", 500.0};
11
12     // create derived-class object
13     SalariedCommissionEmployee salariedCommission{
14        "Bob Lewis", 300.0, 5000.0, .04};
15
16     // output objects using static binding
```

```
17      std::cout << fmt::format("{}\n{}:\n{}\n{}\n",
18         "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND",
19         "DERIVED-CLASS OBJECTS WITH STATIC BINDING",
20         salaried.toString(), // static binding
21         salariedCommission.toString()); // static binding
22
23      std::cout << "INVOKING TOSTRING FUNCTION ON BASE-CLASS AND\n"
24          << "DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING\n\n";
25
26      // natural: aim base-class pointer at base-class object
27      SalariedEmployee* salariedPtr{&salaried};
28      std::cout << fmt::format("{}\n{}:\n{}\n",
29         "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
30         "TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY",
31         salariedPtr->toString()); // base-class version
32
33      // natural: aim derived-class pointer at derived-class object
34      SalariedCommissionEmployee* salariedCommissionPtr{&salariedCommission};
35      std::cout << fmt::format("{}\n{}:\n{}\n",
36         "CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER",
37         "TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
38         salariedCommissionPtr->toString()); // derived-class version
39
40      // aim base-class pointer at derived-class object
41      salariedPtr = &salariedCommission;
42
43      // runtime polymorphism: invokes SalariedCommissionEmployee
44      // via base-class pointer to derived-class object
45      std::cout << fmt::format("{}\n{}:\n{}\n",
46         "CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER",
47         "TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY",
48         salariedPtr->toString()); // derived-class version
49   }
```

```
INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH STATIC BINDING:
name: Sue Jones
salary: $500.00

name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04

INVOKING TOSTRING FUNCTION ON BASE-CLASS AND
DERIVED-CLASS OBJECTS WITH DYNAMIC BINDING

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO BASE-CLASS OBJECT INVOKES BASE-CLASS FUNCTIONALITY:
name: Sue Jones
salary: $500.00

CALLING VIRTUAL FUNCTION TOSTRING WITH DERIVED-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
```

```
gross sales: $5000.00
commission rate: 0.04

CALLING VIRTUAL FUNCTION TOSTRING WITH BASE-CLASS POINTER
TO DERIVED-CLASS OBJECT INVOKES DERIVED-CLASS FUNCTIONALITY:
name: Bob Lewis
salary: $300.00
gross sales: $5000.00
commission rate: 0.04
```

Fig. 10.10 Demonstrating polymorphism by invoking a derived-class `virtual` function via a base-class pointer to a derived-class object.

to **include the `virtual` keyword**:

```
virtual double earnings() const;
virtual std::string toString() const;
```

In class `SalariedCommissionEmployee`'s header (Fig. 10.4), we modified the `earnings` and `toString` prototypes (lines 19–20)

```
double earnings() const;
std::string toString() const;
```

to **include the `override` keyword** (discussed after Fig. 10.10):

```
double earnings() const override;
std::string toString() const override;
```

`SalariedEmployee`'s `earnings` and `toString` functions are `virtual`, so derived class `SalariedCommissionEmployee`'s versions *override* `SalariedEmployee`'s versions. There were no changes to the `SalariedEmployee` and `SalariedCommissionEmployee` member-function implementations, so we reuse the versions of Figs. 10.2 and 10.5.

Runtime Polymorphic Behavior

Now, if we aim a base-class `SalariedEmployee` pointer at a derived-class `SalariedCommissionEmployee` object and use that pointer to call either `earnings` or `toString`, the derived-class object's function will be invoked **polymorphically**. Figure 10.10 demonstrates this runtime polymorphic behavior. First, lines 10–21 create a `SalariedEmployee` and a `SalariedCommissionEmployee`, then use their object names to show their `toString` results. This will help you confirm the dynamic binding results later in the program. Lines 27–38 show again that

- a `SalariedEmployee` pointer aimed at a `SalariedEmployee` object can be used to invoke `SalariedEmployee` functionality and
- a `SalariedCommissionEmployee` pointer aimed at a `SalariedCommissionEmployee` object can be used to invoke `SalariedCommissionEmployee` functionality.

SE Line 41 aims the base-class pointer `salariedPtr` at the derived-class object `salariedCom-mission`. Line 48 invokes member function `toString` via the base-class pointer. As you can see in the output, the derived-class `salariedCommission` object's `toString` member function is invoked. **Declaring the member function `virtual` and invoking it through a base-class pointer or reference causes the program to determine at execution time which function to invoke based on the object's type.**

When `salariedPtr` points to a `SalariedEmployee` object, class `SalariedEmployee`'s `toString` function is invoked (line 31). When `salariedPtr` points to a `SalariedCommissionEmployee` object, class `SalariedCommissionEmployee`'s `toString` function is invoked (line 48). So, **the same `toString` call via a**

base-class pointer to various objects takes on many forms (in this case, two forms). This is runtime polymorphic behavior.

Do Not Call Virtual Functions from Constructors and Destructors

CG Calling a `virtual` function from a base class's constructor or destructor **invokes the base-class version**, even if the base-class constructor or destructor is called while creating or destroying a derived-class object. This is not the behavior you expect for `virtual` functions, so the C++ Core Guidelines recommend that you do not call them from constructors or destructors.[10]

10. "C.82: Don't Call Virtual Functions in Constructors and Destructors." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-ctor-virtual.

11 C++11 `override` Keyword

Err Declaring `SalariedCommissionEmployee`'s `earnings` and `toString` functions using the `override` keyword tells the compiler to check whether the base class has a `virtual` member function with the **same signature**. If not, the compiler generates an error. This ensures that you override the appropriate base-class function. **It also prevents you from accidentally hiding a base-class function with the same name but a different signature**. So, to help you prevent errors, **apply `override` to the prototype of every derived-class function that overrides a `virtual` base-class function**.

CG The C++ Core Guidelines state that

- `virtual` specifically introduces a new `virtual` function in a hierarchy and
- `override` specifically indicates that a derived-class function overrides a base-class `virtual` function.

So, you should use only `virtual` or `override` in each `virtual` function's prototype.[11]

11. "C.128: Virtual Functions Should Specify Exactly One of `virtual`, `override`, or `final`." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-override.

10.7.5 `virtual` Destructors

CG SE The C++ Core Guidelines recommend including a **`virtual` destructor** in *every* class that contains `virtual` functions.[12] This helps prevent subtle errors when a derived class has a custom destructor. **In a class that does not have a destructor, the compiler generates one for you, but the generated one is not `virtual`**. So, in Modern C++, the `virtual` destructor definition for most classes is written as

12. "C.35: A Base Class Destructor Should Be Either `public` and `virtual`, or `protected` and Non-`virtual`." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-virtual.

```
virtual ~SalariedEmployee() = default;
```

This enables you to declare the destructor `virtual` and still have the compiler generate a default destructor for the class—via the notation `= default`.

Before C++11, a derived class could override any base-class `virtual` function. In C++11 and beyond, a `virtual` function that's declared `final` in its prototype, as in
returnType someFunction (*parameters*) `final;`

SE *cannot* be overridden in any derived class. **In a multilevel class hierarchy, this guarantees that the `final` member-function definition will be used by all subsequent direct and indirect derived classes.**

Similarly, before C++11, any existing class could be used as a base class at any point in a hierarchy. In C++11 and beyond, you can declare a class `final` to prevent it from being used as a base class, as in

```
class MyClass final {
   // class body
};
```

or

```
class DerivedClass : public BaseClass final {
    // class body
 };
```

Err Attempting to override a `final` member function or inherit from a `final` base class results in a compilation error.

Perf A benefit of declaring a `virtual` function `final` is that once the compiler knows a `virtual` function cannot be overridden, it can perform various optimizations. For instance, the compiler might be able to determine at compile-time the correct function to call.[13] This optimization is called **devirtualization**.[14]

13. Matt Godbolt, "Optimizations in C++ Compilers," November 12, 2019. Accessed January 11, 2022. https://queue.acm.org/detail.cfm?id=3372264.

14. Although the C++ standard document discusses a number of possible optimizations, it does not require them. They happen at the discretion of the compiler implementers.

There are cases in which the compiler can devirtualize `virtual` function calls even if the `virtual` functions are not declared `final`. For example, sometimes the compiler can recognize at compile-time the type of object that will be used at runtime. In this case, a call to a non-`final` `virtual` function can be bound at compile-time.[15,16]

15. Godbolt, "Optimizations in C++ Compilers."

16. Sy Brand, "The Performance Benefits of Final Classes," March 2, 2020. Accessed January 11, 2022. https://devblogs.microsoft.com/cppblog/the-performance-benefits-of-final-classes/.

10.8 Abstract Classes and Pure `virtual` Functions

SE **There are cases in which it's useful to define classes from which you never intend to instantiate any objects. Such an abstract class defines a common `public` interface for the classes derived from it in a class hierarchy.** Because abstract classes are used as base classes in inheritance

hierarchies, we refer to them as **abstract base classes**. Such classes cannot be used to create objects because, as we'll see, abstract classes are missing pieces. Derived classes must define the "missing pieces" before derived-class objects can be instantiated. We build programs with abstract classes in Sections 10.9, 10.12 and 10.14.

Classes that can be used to instantiate objects are called **concrete classes**. Such classes define or inherit implementations of *every* member function they or their base classes declare. A good example of an abstract class is base class `Shape` in Section 10.2's shape hierarchy. We then have an abstract base class `TwoDimensionalShape` with derived concrete classes `Circle`, `Square` and `Triangle`. We also have an abstract base class, `ThreeDimensionalShape`, with derived concrete classes `Cube`, `Sphere` and `Tetrahedron`. **Abstract base classes are too general to define objects.** For example, if someone tells you to "draw the two-dimensional shape," what shape would you draw? **Concrete classes provide the specifics that make it possible to instantiate objects.**

10.8.1 Pure `virtual` Functions

A class is made abstract by declaring one or more pure `virtual` functions, each specified by placing "`= 0`" in its function prototype, as in

```
virtual void draw() const = 0; // pure virtual function
```

The "`= 0`" is a **pure specifier**. Pure `virtual` functions do not provide implementations. **Each concrete derived class must override its base class's pure `virtual` functions with concrete implementations.** Otherwise, the derived class is also abstract. By contrast, a regular `virtual` function has an implementation, giving the derived class the *option* of overriding the function or simply inheriting the base class's implementation. An abstract class also can have data members and concrete functions. As you'll see in Section 10.12, **an abstract class with all pure `virtual` functions is sometimes called a pure abstract class or an interface**.

A pure `virtual` function is used when the base class does not know how to implement a function, but all concrete derived classes should implement it. Returning to our earlier `SpaceObject`s example, it does not make sense for the base class `SpaceObject` to implement a `draw` function. There's no way to draw a generic space object without knowing which specific type of space object is being drawn.

Although we cannot instantiate objects of an abstract base class, **we can declare pointers and references of the abstract-base-class type.** These can refer to objects of any concrete classes derived from the abstract base class. Programs typically use such pointers and references to manipulate derived-class objects polymorphically at runtime.

10.8.2 Device Drivers: Polymorphism in Operating Systems

Polymorphism is particularly effective for implementing layered software systems. For example, in operating systems, each type of physical input/output device could operate quite differently from the others. Even so, commands to read or write data from and to devices, respectively, may have a certain uniformity. The write message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a

given type. However, the write call itself is no different from the write to any other device in the system—place some bytes from memory onto that device.

Consider an object-oriented operating system that uses an abstract base class to provide an interface appropriate for all device drivers. Through inheritance from that abstract base class, derived classes are formed that all operate similarly. The `public` functions offered by the device drivers are pure `virtual` functions in the abstract base class. The implementations of these pure `virtual` functions are provided in the derived classes that correspond to the specific types of device drivers.

SE This architecture also allows new devices to be added to a system easily. The user can just plug in the device and install its new device driver. The operating system "talks" to this new device through its device driver, which has the same public member functions as all other device drivers—those defined in the device-driver abstract base class.

10.9 Case Study: Payroll System Using Runtime Polymorphism

Let's use an abstract class and runtime polymorphism to perform payroll calculations for two employee types. We create an `Employee` hierarchy to solve the following problem:
A company pays its employees weekly. The employees are of two types:

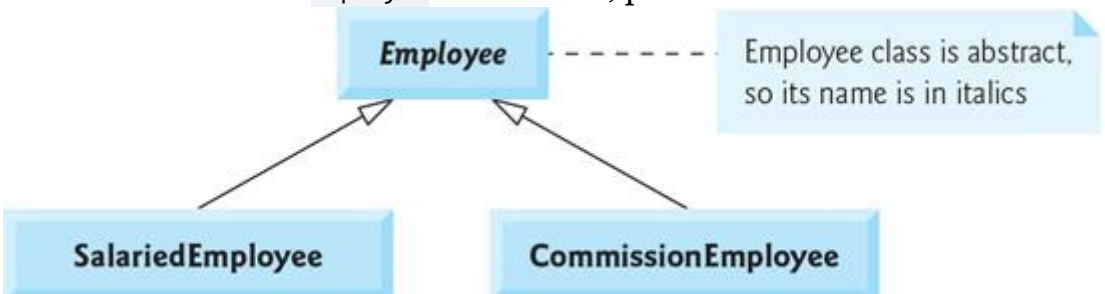
1. Salaried employees are paid a fixed salary regardless of their hours worked.

2. Commission employees are paid a percentage of their sales.

The company wants to implement a C++ program that performs its payroll calculations polymorphically.

SE **Many hierarchies start with an abstract base class followed by a row of derived classes that are `final`.** We use abstract class `Employee` to represent the **general concept of an employee** and define the **"interface" to the hierarchy—that is, the member functions all `Employee` derived classes must have, so a program can invoke them on all `Employee` objects**. Also, regardless of how earnings are calculated, each employee has a name. So, we'll define a `private` data member `m_name` in abstract base class `Employee`.

The `final` classes that we derive directly from `Employee` are `SalariedEmployee` and `CommissionEmployee`. Each is a **leaf node** in the class hierarchy and cannot be a base class. The following UML class diagram shows the inheritance hierarchy for our runtime polymorphic payroll application. The abstract class name `Employee` is *italicized*, per the UML's convention.



SE A derived class can inherit interface and/or implementation from a base class. **Hierarchies designed for interface inheritance tend to have concrete definitions of their functionality lower in the hierarchy.** A base class declares one or more functions that should be defined by every derived class, but the individual derived classes provide the implementations of the function(s).

The following subsections implement the `Employee` class hierarchy. The first three each define one of the abstract or concrete classes. The last contains a test program that builds concrete-class objects and processes them with runtime polymorphism.

10.9.1 Creating Abstract Base Class `Employee`

Class `Employee` (Figs. 10.11–10.12, discussed in further detail shortly) provides functions `earnings` and `toString`, and *get* and *set* functions that manipulate `Employee`'s `m_name` data member. An `earnings` function certainly applies generally to all `Employee`s, but each earnings calculation depends on its class. So we declare `earnings` as pure `virtual` in base class `Employee` because a default implementation does not make sense for that function. There's not enough information to determine what amount `earnings` should return.

Each derived class overrides `earnings` with an appropriate implementation. To calculate an employee's earnings, the program assigns an employee object's address to a base-class `Employee` pointer, then invokes the object's `earnings` function.

SE The test program maintains a `vector` of `Employee` pointers, each of which points to an object that *is an* `Employee`—specifically, any object of a concrete derived class of `Employee`. The program iterates through the `vector` and calls each `Employee`'s `earnings` function. C++ processes these calls polymorphically. **Including `earnings` as a pure `virtual` function in `Employee` forces every derived class of `Employee` that wishes to be a concrete class to override `earnings`.**

`Employee`'s `toString` function returns a `string` containing the `Employee`'s name. Each class derived from `Employee` will override `toString` to return the name followed by the rest of the `Employee`'s information. Each derived class's `toString` could also call `earnings`, even though `earnings` is a pure `virtual` function in `Employee`. Each concrete class is guaranteed to have an `earnings` implementation. Even class `Employee`'s `toString` function can call `earnings`. When you call `toString` through an `Employee` pointer or reference at runtime, you're always calling it on a concrete derived-class object.

The following diagram shows the hierarchy's three classes down the left and functions `earnings` and `toString` across the top. For each class, the diagram shows the desired return value of each function.

	earnings	toString
Employee	pure virtual	name: *m_name*
Salaried-Employee	*m_salary*	name: *m_name* salary: *m_salary*
Commission-Employee	*m_commissionRate* * *m_grossSales*	name: *m_name* gross sales: *m_grossSales* commission rate: *m_commissionRate*

Italic text represents where the values from a particular object are used in the `earnings` and `toString` functions. Class `Employee` specifies "pure `virtual`" for function `earnings` to indicate it does not provide an implementation. Each derived class overrides this function to provide an appropriate implementation. We do not list base class `Employee`'s *get* and *set* functions because the derived classes do not override them. Each function is inherited and used "as is" by the derived classes.

Employee Class Header

Consider class `Employee`'s header (Fig. 10.11). Its `public` member functions include

- a constructor that takes the name as an argument (line 9),
- 11 a C++11 `default`ed virtual destructor (line 10),
- a *set* function to set the name (line 12),
- a *get* function to return the name (line 13),
- pure `virtual` function `earnings` (line 16) and
- `virtual` function `toString` (line 17).

```
1  // Fig. 10.11: Employee.h
2  // Employee abstract base class.
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5  #include <string_view>
6
7  class Employee {
8  public:
9     explicit Employee(std::string_view name);
10    virtual ~Employee() = default; // compiler generates virtual destructor
11
12    void setName(std::string_view name);
13    std::string getName() const;
14
15    // pure virtual function makes Employee an abstract base class
16    virtual double earnings() const = 0; // pure virtual
17    virtual std::string toString() const; // virtual
18 private:
19    std::string m_name;
20 };
```

Fig. 10.11 `Employee` abstract base class.

Recall that we declared `earnings` as a pure `virtual` function because first we must know the specific `Employee` type to determine the appropriate `earnings` calculation. Each concrete derived class must provide an `earnings` implementation. Then, using a base-class `Employee` pointer or reference, a program can invoke function `earnings` polymorphically for an object of any concrete derived class of `Employee`.

Employee Class Member-Function Definitions

Figure 10.12 contains `Employee`'s member-function definitions. No implementation is provided for `virtual` function `earnings`. The `virtual` function `toString` implementation (lines 17–19) will be overridden in each derived class. Those derived-class `toString` functions will call `Employee`'s `toString` to get a `string` containing the information common to all classes in the `Employee` hierarchy (i.e., the name).

```
1  // Fig. 10.12: Employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <fmt/format.h>
5  #include "Employee.h" // Employee class definition
6
7  // constructor
8  Employee::Employee(std::string_view name) : m_name{name} {} // empty body
9
10 // set name
11 void Employee::setName(std::string_view name) {m_name = name;}
12
13 // get name
14 std::string Employee::getName() const {return m_name;}
15
16 // return string representation of an Employee
17 std::string Employee::toString() const {
18    return fmt::format("name: {}", getName());
19 }
```

Fig. 10.12 `Employee` class implementation file.

10.9.2 Creating Concrete Derived Class `SalariedEmployee`

Class `SalariedEmployee` (Figs. 10.13–10.14) derives from class `Employee` (Fig. 10.13, line 8). `SalariedEmployee`'s `public` member functions include

- a constructor that takes a name and a salary as arguments (line 10),
- 11 a C++11 `default virtual` destructor (line 11),
- a *set* function to assign a new non-negative value to data member `m_salary` (line 13) and a *get* function to return `m_salary`'s value (line 14),
- an `override` of `Employee`'s `virtual` function `earnings` that calculates a `SalariedEmployee`'s earnings (line 17) and
- an `override` of `Employee`'s `virtual` function `toString` (line 18) that returns a `SalariedEmployee`'s string representation.

```
1  // Fig. 10.13: SalariedEmployee.h
2  // SalariedEmployee class derived from Employee.
3  #pragma once
4  #include <string> // C++ standard string class
5  #include <string_view>
6  #include "Employee.h" // Employee class definition
7
8  class SalariedEmployee final : public Employee {
9  public:
10    SalariedEmployee(std::string_view name, double salary);
11    virtual ~SalariedEmployee() = default; // virtual destructor
12
13    void setSalary(double salary);
14    double getSalary() const;
15
16    // keyword override signals intent to override
```

```
17      double earnings() const override; // calculate earnings
18      std::string toString() const override; // string representation
19   private:
20      double m_salary{0.0};
21   };
```

Fig. 10.13 `SalariedEmployee` class header.

SalariedEmployee Class Member-Function Definitions

Figure 10.14 defines `SalariedEmployee`'s member functions:

- Its constructor passes the `name` argument to the `Employee` constructor (line 9) to initialize the inherited `private` data member that's not directly accessible in the derived class.
- Function `earnings` (line 27) overrides `Employee`'s pure `virtual` function `earnings` to provide a concrete implementation that returns the weekly salary. If we did not override `earnings`, `SalariedEmployee` would inherit `Employee`'s pure `virtual earnings` function and be abstract.
- `SalariedEmployee`'s `toString` function (lines 30–33) overrides `Employee`'s `toString`. If it did not, the class would inherit `Employee`'s, which returns a `string` containing the employee's name. `SalariedEmployee`'s `toString` returns a `string` containing the `Employee::toString()` result and the `SalariedEmployee`'s salary.

`SalariedEmployee`'s header declared member functions `earnings` and `toString` as `override` to ensure that we correctly override them. Recall that these are `virtual` in base class `Employee`, so they remain `virtual` throughout the class hierarchy.

```
1   // Fig. 10.14: SalariedEmployee.cpp
2   // SalariedEmployee class member-function definitions.
3   #include <fmt/format.h>
4   #include <stdexcept>
5   #include "SalariedEmployee.h" // SalariedEmployee class definition
6
7   // constructor
8   SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
9      : Employee{name} {
10     setSalary(salary);
11  }
12
13  // set salary
14  void SalariedEmployee::setSalary(double salary) {
15     if (salary < 0.0) {
16        throw std::invalid_argument("Weekly salary must be >= 0.0");
17     }
18
19     m_salary = salary;
20  }
21
22  // return salary
23  double SalariedEmployee::getSalary() const {return m_salary;}
24
25  // calculate earnings;
26  // override pure virtual function earnings in Employee
27  double SalariedEmployee::earnings() const {return getSalary();}
28
```

```
29   // return a string representation of SalariedEmployee
30   std::string SalariedEmployee::toString() const {
31      return fmt::format("{}\n{}: ${:.2f}", Employee::toString(),
32         "salary", getSalary());
33   }
```

Fig. 10.14 `SalariedEmployee` class implementation file.

10.9.3 Creating Concrete Derived Class `CommissionEmployee`

Class `CommissionEmployee` (Figs. 10.15–10.16) derives from `Employee` (Fig. 10.15, line 8). The member-function implementations in Fig. 10.16 include

- a constructor (lines 8–12) that takes a name, gross sales and commission rate, then passes the name to `Employee`'s constructor (line 9) to initialize the inherited data members,
- *set* functions (lines 15–21 and 27–34) to assign new values to data members `m_grossSales` and `m_commissionRate`,
- *get* functions (lines 24 and 37–39) to return the values of `m_grossSales` and `m_commissionRate`,
- an `override` of `Employee`'s `earnings` function (lines 42–44) that calculates a `CommissionEmployee`'s earnings and
- an `override` of `Employee`'s `toString` function (lines 47–51) to return a `string` containing the `Employee::toString()` result, the gross sales and the commission rate.

```
1   // Fig. 10.15: CommissionEmployee.h
2   // CommissionEmployee class derived from Employee.
3   #pragma once
4   #include <string>
5   #include <string_view>
6   #include "Employee.h" // Employee class definition
7
8   class CommissionEmployee final : public Employee {
9   public:
10     CommissionEmployee(std::string_view name, double grossSales,
11        double commissionRate);
12     virtual ~CommissionEmployee() = default; // virtual destructor
13
14     void setGrossSales(double grossSales);
15     double getGrossSales() const;
16
17     void setCommissionRate(double commissionRate);
18     double getCommissionRate() const;
19
20     // keyword override signals intent to override
21     double earnings() const override; // calculate earnings
22     std::string toString() const override; // string representation
23  private:
24     double m_grossSales{0.0};
25     double m_commissionRate{0.0};
26  };
```

Fig. 10.15 `CommissionEmployee` class header.

```
1   // Fig. 10.16: CommissionEmployee.cpp
2   // CommissionEmployee class member-function definitions.
```

```
3   #include <fmt/format.h>
4   #include <stdexcept>
5   #include "CommissionEmployee.h" // CommissionEmployee class definition
6
7   // constructor
8   CommissionEmployee::CommissionEmployee(std::string_view name,
9      double grossSales, double commissionRate) : Employee{name} {
10     setGrossSales(grossSales);
11     setCommissionRate(commissionRate);
12  }
13
14  // set gross sales amount
15  void CommissionEmployee::setGrossSales(double grossSales) {
16     if (grossSales < 0.0) {
17        throw std::invalid_argument("Gross sales must be >= 0.0");
18     }
19
20     m_grossSales = grossSales;
21  }
22
23  // return gross sales amount
24  double CommissionEmployee::getGrossSales() const {return m_grossSales;}
25
26  // set commission rate
27  void CommissionEmployee::setCommissionRate(double commissionRate) {
28     if (commissionRate <= 0.0 || commissionRate >= 1.0) {
29        throw std::invalid_argument(
30           "Commission rate must be > 0.0 and < 1.0");
31     }
32
33     m_commissionRate = commissionRate;
34  }
35
36  // return commission rate
37  double CommissionEmployee::getCommissionRate() const {
38     return m_commissionRate;
39  }
40
41  // calculate earnings
42  double CommissionEmployee::earnings() const {
43     return getGrossSales() * getCommissionRate();
44  }
45
46  // return string representation of CommissionEmployee object
47  std::string CommissionEmployee::toString() const {
48     return fmt::format("{}\n{}: ${:.2f}\n{}: {:.2f}", Employee::toString(),
49        "gross sales", getGrossSales(),
50        "commission rate", getCommissionRate());
51  }
```

Fig. 10.16 `CommissionEmployee` class implementation file.

10.9.4 Demonstrating Runtime Polymorphic Processing

To test our `Employee` hierarchy, the program in Fig. 10.17 creates an object of each concrete class—`SalariedEmployee` and `CommissionEmployee`. The program manipulates these objects first via their object

names, then with runtime polymorphism, using a `vector` of `Employee` base-class pointers. Lines 16–17 create objects of each concrete derived class. Lines 20–23 output each employee's information and earnings. **These lines use the variable names for each object, so the compiler can identify at compile-time each object's type to determine which `toString` and `earnings` functions to call.**

```
1  // fig10_17.cpp
2  // Processing Employee derived-class objects with variable-name handles
3  // then polymorphically using base-class pointers and references
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <vector>
7  #include "Employee.h"
8  #include "SalariedEmployee.h"
9  #include "CommissionEmployee.h"
10
11 void virtualViaPointer(const Employee* baseClassPtr); // prototype
12 void virtualViaReference(const Employee& baseClassRef); // prototype
13
14 int main() {
15    // create derived-class objects
16    SalariedEmployee salaried{"John Smith", 800.0};
17    CommissionEmployee commission{"Sue Jones", 10000, .06};
18
19    // output each Employee
20    std::cout << "EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES\n"
21       << fmt::format("{}\n{}{:.2f}\n\n{}\n{}{:.2f}\n\n",
22             salaried.toString(), "earned $", salaried.earnings(),
23             commission.toString(), "earned $", commission.earnings());
24
25    // create and initialize vector of base-class pointers
26    std::vector<Employee*> employees{&salaried, &commission};
27
28    std::cout << "EMPLOYEES PROCESSED POLYMORPHICALLY VIA"
29       << " DYNAMIC BINDING\n\n";
30
31    // call virtualViaPointer to print each Employee
32    // and earnings using dynamic binding
33    std::cout << "VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS\n";
34
35    for (const Employee* employeePtr : employees) {
36       virtualViaPointer(employeePtr);
37    }
38
39    // call virtualViaReference to print each Employee
40    // and earnings using dynamic binding
41    std::cout << "VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES\n";
42
43    for (const Employee* employeePtr : employees) {
44       virtualViaReference(*employeePtr); // note dereferenced pointer
45    }
46 }
47
48 // call Employee virtual functions toString and earnings via a
```

```
49   // base-class pointer using dynamic binding
50   void virtualViaPointer(const Employee* baseClassPtr) {
51      std::cout << fmt::format("{}\nearned ${:.2f}\n\n",
52         baseClassPtr->toString(), baseClassPtr->earnings());
53   }
54
55   // call Employee virtual functions toString and earnings via a
56   // base-class reference using dynamic binding
57   void virtualViaReference(const Employee& baseClassRef) {
58      std::cout << fmt::format("{}\nearned ${:.2f}\n\n",
59         baseClassRef.toString(), baseClassRef.earnings());
60   }
```

```
EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00
```

Fig. 10.17 Processing `Employee` derived-class objects with variable-name handles then polymorphically using base-class pointers and references.

Creating a `vector` of Employee Pointers

Line 26 creates and initializes the `vector employees`, containing two `Employee` pointers aimed at the objects `salaried` and `commission`, respectively. The compiler allows the elements to be initialized with these objects' addresses because a `SalariedEmployee` *is an* `Employee` and a `CommissionEmployee` *is an* `Employee`.

Function `virtualViaPointer`

SE Lines 35–37 traverse the `vector employees` and invoke function `virtualViaPointer` (lines 50–53) with each element as an argument. Function `virtualViaPointer` receives in its parameter `baseClassPtr` the address stored in a given `employees` element, then uses the pointer to invoke `virtual` functions `toString` and `earnings`. The function does not contain any `SalariedEmployee` or `CommissionEmployee` type information—it knows only about base class `Employee`. The program repeatedly aims `baseClassPtr` at different concrete derived-class objects, so **the compiler cannot know which concrete class's functions to call through `baseClassPtr`—it must resolve these calls at runtime using dynamic binding.** At execution time, each `virtual`-function call correctly invokes the function on the object to which `baseClassPtr` currently points. The output shows that each class's appropriate functions are invoked and displayed the correct information. Obtaining each `Employee`'s `earnings` via runtime polymorphism produces the same results as lines 22 and 23.

Function `virtualViaReference`

Lines 43–45 traverse `employees` and invoke function `virtualViaReference` (lines 57–60) with each element as an argument. Function `virtualViaReference` receives in its parameter `baseClassRef` (of type `const Employee&`) a reference to the object obtained by dereferencing the pointer stored in an element of `vector employees` (line 44). Each call to this function invokes `virtual` functions `toString` and `earnings` via `baseClassRef` to demonstrate that runtime polymorphic processing also occurs with base-class references. Each `virtual` function invocation calls the function on the object `baseClassRef` references at runtime. This is another example of dynamic binding. The output produced using base-class references is identical to that produced using base-class pointers and via static binding earlier in the program.

10.10 Runtime Polymorphism, Virtual Functions and Dynamic Binding "Under the Hood"

Perf Let's consider how C++ can implement runtime polymorphism, `virtual` functions and dynamic binding. This will give you a solid understanding of how these capabilities can work. More importantly, you'll appreciate the overhead of runtime polymorphism—in terms of additional memory consumption and processor time. This can help you determine when to use runtime polymorphism and when to avoid it. **C++ standard-library classes generally are implemented without `virtual` functions to avoid the associated execution-time overhead and achieve optimal performance.**

First, we'll explain the data structures that the compiler can build at compile-time to support polymorphism at execution time. You'll see that this can be accomplished through three levels of pointers—that is, *triple indirection*. Then we'll show how an executing program can use these data structures to execute `virtual` functions and achieve the dynamic binding associated with polymorphism. Our discussion explains a possible implementation.

Virtual-Function Tables

When C++ compiles a class with one or more `virtual` functions, it builds a **virtual function table (vtable)** for that class. The *vtable* contains pointers to the class's `virtual` functions. A **pointer to a function** contains the starting address in memory of the code that performs the function's task. Just as an array name is implicitly convertible to the address of the array's first element, a function name is implicitly convertible to the starting address of its code.

With dynamic binding, an executing program uses a class's *vtable* to select the proper function implementation each time a `virtual` function is called on an object of that class. The leftmost column of Fig. 10.18 illustrates the *vtables* for `Employee`, `SalariedEmployee` and `CommissionEmployee`.

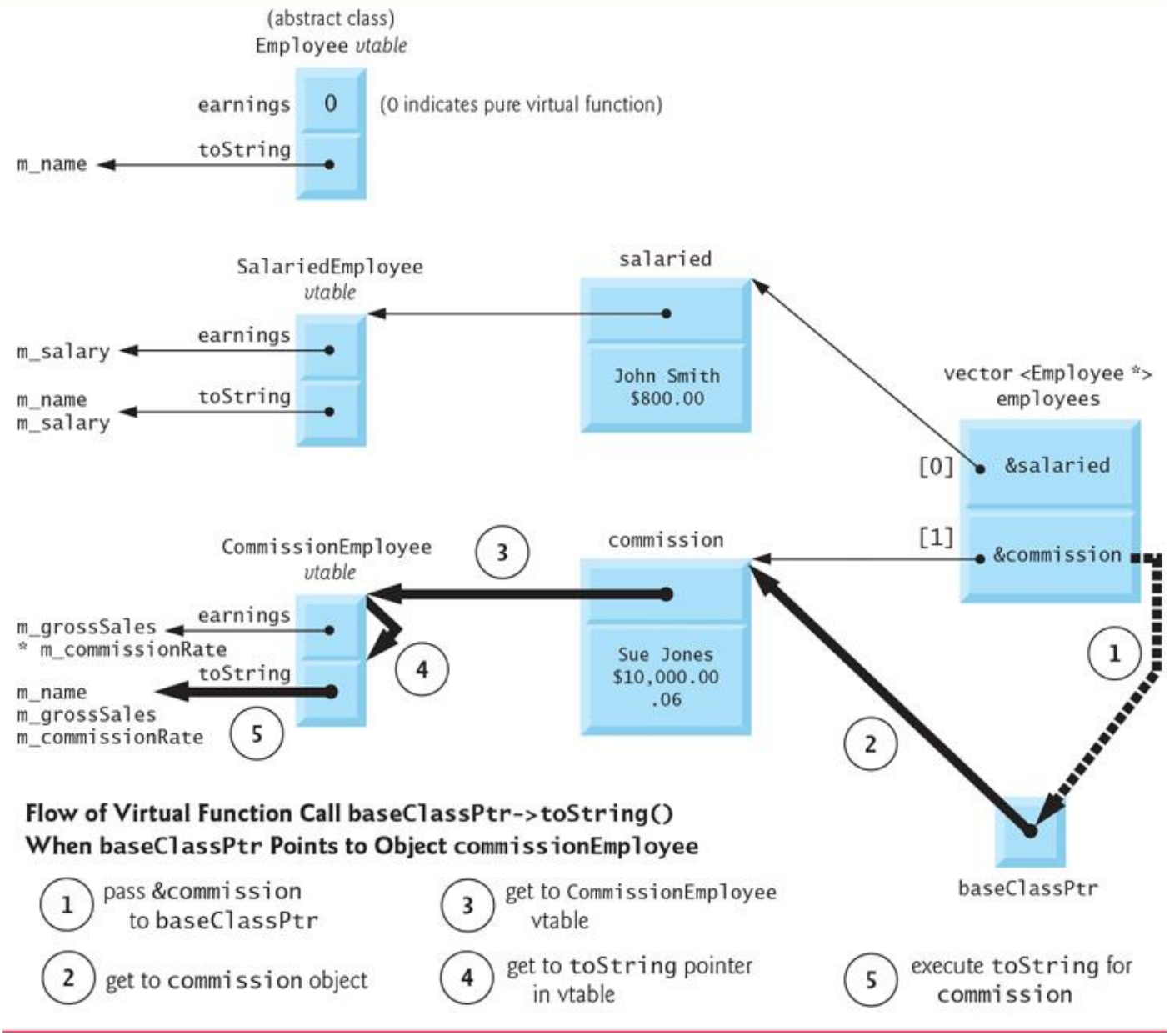


Fig. 10.18 How `virtual` function calls work.

Employee Class vtable

The first function pointer in the `Employee` class *vtable* is set to `0` (i.e., `nullptr`) because `earnings` is a pure `virtual` function with no implementation. The second points to `toString`, which returns a `string` containing the employee's name. We've abbreviated each `toString` function's output in this figure to conserve space. **Any class with one or more pure `virtual` functions (represented with the value `0`) in its vtable is an abstract class.** `SalariedEmployee` and `CommissionEmployee` are concrete classes—they have no `nullptr`s in their *vtables*.

`SalariedEmployee` Class vtable

The `earnings` function pointer in the `SalariedEmployee` *vtable* points to that class's override of the `earnings` function, which returns the salary. `SalariedEmployee` also overrides `toString`, so the corresponding function pointer points to `SalariedEmployee`'s `toString` function, which returns the employee's name and salary.

`CommissionEmployee` Class vtable

The `earnings` function pointer in the `CommissionEmployee` *vtable* points to the `CommissionEmployee`'s `earnings` function, which returns the employee's gross sales multiplied by the commission rate. The `toString` function pointer points to the `CommissionEmployee` version of the function, which returns the employee's name, commission rate and gross sales. As in class `SalariedEmployee`, both functions override class `Employee`'s functions.

Inheriting Concrete `virtual` Functions

Each concrete class in our `Employee` case study provides `virtual earnings` and `toString` implementations. You've learned that because `earnings` is a pure `virtual` function, each *direct* derived class of `Employee` must implement `earnings` to be a concrete class. Direct derived classes do not need to implement `toString` to be considered concrete—they can inherit class `Employee`'s `toString` implementation. In our case, both derived classes override `Employee`'s `toString`.

If a derived class in our hierarchy were to inherit `toString` and not override it, this function's vtable pointer would simply point to the inherited implementation. For example, if `CommissionEmployee` did not override `toString`, `CommissionEmployee`'s `toString` function pointer in the *vtable* would point to the same `toString` function as in class `Employee`'s *vtable*.

Three Levels of Pointers to Implement Runtime Polymorphism

Runtime polymorphism can be accomplished through an elegant data structure involving three levels of pointers. We've discussed one level—the function pointers in the *vtable*. These point to the functions that execute when a `virtual` function is invoked.

Now we consider the second level of pointers. **Whenever an object with one or more `virtual` functions is instantiated, the compiler attaches to it a pointer to the class's vtable.** This pointer is usually at the front of the object, but it isn't required to be implemented that way. In the *vtable* diagram, these pointers are associated with the `Salaried-Employee` and `CommissionEmployee` objects defined in Fig. 10.17. The diagram shows each object's data member values.

The third level of pointers contains the addresses of the objects on which the `virtual` functions will be called. The *vtable* diagram's rightmost column depicts the `vector employees` containing these `Employee` pointers.

Now let's see how a typical `virtual` function call executes. Consider in the function `virtualViaPointer` the call `baseClassPtr->toString()` (Fig. 10.17, line 52). Assume that `baseClassPtr` contains `employees[1]`, `commission`'s address in `employees`. When this statement is compiled, the compiler sees that the call is made via a base-class pointer and that `toString` is a `virtual` function. The compiler sees that `toString` is the second entry in each *vtable*. So it includes an **offset** into the table of machine-language object-code pointers to find the code that will execute the `virtual` function call.

The compiler generates code that performs the following operations—the numbers in the list correspond to the circled numbers in Fig. 10.18:

1. Select the *i*th `employees` entry—the `commission` object's address—and pass it as an argument to function `virtualViaPointer`. This aims the `baseClassPtr` parameter at the `commission` object.
2. Dereference that pointer to get to the `commission` object, which begins with a pointer to class `CommissionEmployee`'s *vtable*.
3. Dereference the *vtable* pointer to get to class `CommissionEmployee`'s *vtable*.
4. Skip the offset to select the `toString` function pointer.[17]

17. In practice, *Steps 3* and *4* can be implemented as a single machine instruction.

5. Execute `toString` for the `commission` object, returning a `string` containing the employee's name, gross sales and commission rate.

Perf The *vtable* diagram's data structures may appear complex. **The compiler manages this complexity and hides it from you**, making runtime polymorphic programming straightforward. The pointer dereferencing operations and memory accesses for each `virtual` function call require additional execution time. The *vtables* and *vtable* pointers added to the objects require some additional memory.

Perf **Runtime polymorphism, as typically implemented with `virtual` functions and dynamic binding in C++, is efficient.** For most applications, you can use these capabilities with nominal impact on execution performance and memory consumption. In some situations, polymorphism's overhead may be too high—such as in real-time applications with stringent execution-timing performance requirements or in an application with an enormous number of *small* objects in which the size of the *vtable* pointer is large when compared to each object's size.

10.11 Non-Virtual Interface (NVI) Idiom

The **non-virtual interface (NVI) idiom**[18,19] is another way to implement runtime polymorphism using class hierarchies. Herb Sutter first proposed NVI in his "Virtuality" paper.[20] Sutter lists four guidelines for implementing class hierarchies, each of which we'll use in this example:

18. "Non-Virtual Interface (NVI) Pattern." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Non-virtual_interface_pattern.

19. Marius Bancila, *Modern C++ Programming Cookbook: Master C++ Core Language and Standard Library Features, with over 100 Recipes, Updated to C++20*. Birmingham: Packt Publishing, 2020, pp. 562–567.

20. Herb Sutter, "Virtuality," *C/C++ Users Journal*, vol. 19, no. 9, September 2001. Accessed January 11, 2022. http://www.gotw.ca/publications/mill18.htm.

1. "Prefer to make interfaces non-`virtual`, using Template Method"—an object-oriented design pattern.[21,22] Sutter explains that a `public virtual` function serves *two* purposes—it describes part of a class's interface and enables derived classes to customize behavior by overriding the `virtual` function. He recommends having each function serve only *one* purpose.

21. "Template Method Pattern." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Template_method_pattern.

22. Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995, pp. 325–330.

2. "Prefer to make `virtual` functions `private`." **A derived class can override its base class's `private virtual` functions.** Making a `virtual` function `private` ensures that it serves one purpose—enabling derived classes to customize behavior by overriding the `virtual` function. A non-`virtual` function in the base class invokes the `private virtual` function internally as an implementation detail.

3. "Only if derived classes need to invoke the base implementation of a `virtual` function, make the `virtual` function `protected`." This enables derived classes to override the base-class `virtual` function and take advantage of its base-class implementation to avoid duplicating code in the derived class.

4. "A base-class destructor should be either `public` and `virtual`, or `protected` and non-`virtual`." Here, we'll again make the `Employee` base class's destructor `public` and `virtual`. In Chapter 11, you'll see

that `public virtual` destructors are important when deleting dynamically allocated derived-class objects via base-class pointers.

SE Sutter's paper demonstrates each recommendation, focusing on separating a class's interface from its implementation as a good software-engineering practice.[23]

23. For some related inheritance discussions, see the `isocpp.org` FAQ, "Inheritance—What Your Mother Never Told You" at https://isocpp.org/wiki/faq/strange-inheritance.

Refactoring Class `Employee` for the NVI Idiom

Per Sutter's guidelines, let's refactor our `Employee` class (Figs. 10.19–10.20). Most of the code is identical to Section 10.9, so we focus here on only the changes. There are several key changes in Fig. 10.19:

- `Employee`'s `public earnings` (line 15) and `toString` (line 16) member functions are no longer declared `virtual`, and, as you'll see, `earnings` will have an implementation.

Functions `earnings` and `toString` each now serve *one* purpose—to allow client code to get an `Employee`'s earnings and `string` representation, respectively. Their original second purpose as **customization points for derived classes** will now be implemented via new member functions.

- **We added the `protected virtual` function `getString` (line 18) to serve as a customization point for derived classes.** This function is called by the non-`virtual toString` function. **A base class's `protected` members are accessible to its derived classes.** As you'll see, our derived classes' `getString` functions will both override and call class `Employee`'s `getString`.
- **We added the `private` pure `virtual` function `getPay` (line 21) to serve as a customization point for derived classes.** This function is called by the non-`virtual earnings` function. Derived classes override `getPay` to specify custom earnings calculations.

```
1  // Fig. 10.19: Employee.h
2  // Employee abstract base class.
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5  #include <string_view>
6
7  class Employee {
8  public:
9     Employee(std::string_view name);
10    virtual ~Employee() = default;
11
12    void setName(std::string_view name);
13    std::string getName() const;
14
15    double earnings() const; // not virtual
16    std::string toString() const; // not virtual
17 protected:
18    virtual std::string getString() const; // virtual
19 private:
20    std::string m_name;
21    virtual double getPay() const = 0; // pure virtual
22 };
```

Fig. 10.19 Employee abstract base class.

There are several key changes in class `Employee`'s member-function implementations (Fig. 10.20):

- Member function `earnings` (line 17) now provides a **concrete implementation** that returns the result of calling the `private` pure `virtual` function `getPay`. This call is allowed because, at execution time, earnings will be called on an object of a concrete derived class of `Employee`.
- Member function `toString` (line 20) now returns the result of calling the `protected virtual` function `getString`.
- We now define the new `protected` member function `getString` to specify an `Employee`'s default `string` representation containing an `Employee`'s name. This function is `protected` so **derived classes can override it and call it to get the base-class part of the derived-class `string` representations**.

```
1  // Fig. 10.20: Employee.cpp
2  // Abstract-base-class Employee member-function definitions.
3  // Note: No definitions are given for pure virtual functions.
4  #include <fmt/format.h>
5  #include "Employee.h" // Employee class definition
6
7  // constructor
8  Employee::Employee(std::string_view name) : m_name{name} {} // empty body
9
10 // set name
11 void Employee::setName(std::string_view name) {m_name = name;}
12
13 // get name
14 std::string Employee::getName() const {return m_name;}
15
16 // public non-virtual function; returns Employee's earnings
17 double Employee::earnings() const {return getPay();}
18
19 // public non-virtual function; returns Employee's string representation
20 std::string Employee::toString() const {return getString();}
21
22 // protected virtual function that derived classes can override and call
23 std::string Employee::getString() const {
24    return fmt::format("name: {}", getName());
25 }
```

Fig. 10.20 Abstract-base-class `Employee` member-function definitions.

Updated Class `SalariedEmployee`

Our refactored `SalariedEmployee` class (Figs. 10.21–10.22) has several key changes:

- The class's header (Fig. 10.21) no longer contains prototypes for the `earnings` and `toString` member functions. **These `public` non-`virtual` base-class functions are now inherited from class `Employee`.**
- The class's `private` section now declares overrides for the base class's `private` pure `virtual` function `getPay` and `protected virtual` function `getString` (Fig. 10.21, lines 19–20). **Function `getString` is `private` in `SalariedEmployee` because this class is `final`, so no other classes can derive from it.**
- The class's member-function implementations (Fig. 10.22) now include overridden function `getPay` (line 27) to return the salary and overridden function `get-String` (lines 30–33) to get a `SalariedEmployee`'s `string` representation. Note that `SalariedEmployee`'s `getString` calls class `Employee`'s `protected getString` to get part of the `string` representation.

```cpp
// Fig. 10.21: SalariedEmployee.h
// SalariedEmployee class derived from Employee.
#pragma once
#include <string> // C++ standard string class
#include <string_view>
#include "Employee.h" // Employee class definition

class SalariedEmployee final : public Employee {
public:
   SalariedEmployee(std::string_view name, double salary);
   virtual ~SalariedEmployee() = default; // virtual destructor

   void setSalary(double salary);
   double getSalary() const;
private:
   double m_salary{0.0};

   // keyword override signals intent to override
   double getPay() const override; // calculate earnings
   std::string getString() const override; // string representation
};
```

Fig. 10.21 `SalariedEmployee` class derived from `Employee`.

```cpp
// Fig. 10.22: SalariedEmployee.cpp
// SalariedEmployee class member-function definitions.
#include <fmt/format.h>
#include <stdexcept>
#include "SalariedEmployee.h" // SalariedEmployee class definition

// constructor
SalariedEmployee::SalariedEmployee(std::string_view name, double salary)
   : Employee{name} {
   setSalary(salary);
}

// set salary
void SalariedEmployee::setSalary(double salary) {
   if (salary < 0.0) {
      throw std::invalid_argument("Weekly salary must be >= 0.0");
   }

   m_salary = salary;
}

// return salary
double SalariedEmployee::getSalary() const {return m_salary;}

// calculate earnings;
// override pure virtual function getPay in Employee
double SalariedEmployee::getPay() const {return getSalary();}

// return a string representation of SalariedEmployee
std::string SalariedEmployee::getString() const {
   return fmt::format("{}\n{}: ${:.2f}", Employee::getString(),
```

```
32          "salary", getSalary());
33    }
```

Fig. 10.22 `SalariedEmployee` class member-function definitions.

Updated Class `CommissionEmployee`

Our refactored `CommissionEmployee` class (Figs. 10.23–10.24) has several key changes:

- The class's header (Fig. 10.23) no longer contains prototypes for the `earnings` and `toString` member functions. **These `public` non-`virtual` base-class functions are now inherited from class `Employee`.**
- The class's `private` section now declares overrides for the base class's `private` pure `virtual` function `getPay` and `protected virtual` function `getString` (Fig. 10.23, lines 24–25). **Again, we declared `getString private` because this class is `final`, so no other classes can derive from it.**
- The class's member-function implementations (Fig. 10.24) now include overridden function `getPay` (lines 42–44) to return the commission calculation result and overridden function `getString` (lines 47–51) to get a `CommissionEmployee`'s `string` representation. Function `getString` calls class `Employee`'s `protected getString` to get part of the `string` representation.

```
1   // Fig. 10.23: CommissionEmployee.h
2   // CommissionEmployee class derived from Employee.
3   #pragma once
4   #include <string>
5   #include <string_view>
6   #include "Employee.h" // Employee class definition
7
8   class CommissionEmployee final : public Employee {
9   public:
10     CommissionEmployee(std::string_view name, double grossSales,
11        double commissionRate);
12     virtual ~CommissionEmployee() = default; // virtual destructor
13
14     void setGrossSales(double grossSales);
15     double getGrossSales() const;
16
17     void setCommissionRate(double commissionRate);
18     double getCommissionRate() const;
19  private:
20     double m_grossSales{0.0};
21     double m_commissionRate{0.0};
22
23     // keyword override signals intent to override
24     double getPay() const override; // calculate earnings
25     std::string getString() const override; // string representation
26  };
```

Fig. 10.23 `CommissionEmployee` class derived from `Employee`.

```
1   // Fig. 10.24: CommissionEmployee.cpp
2   // CommissionEmployee class member-function definitions.
3   #include <fmt/format.h>
4   #include <stdexcept>
```

```
5  #include "CommissionEmployee.h" // CommissionEmployee class definition
6
7  // constructor
8  CommissionEmployee::CommissionEmployee(std::string_view name,
9     double grossSales, double commissionRate) : Employee{name} {
10    setGrossSales(grossSales);
11    setCommissionRate(commissionRate);
12 }
13
14 // set gross sales amount
15 void CommissionEmployee::setGrossSales(double grossSales) {
16    if (grossSales < 0.0) {
17       throw std::invalid_argument("Gross sales must be >= 0.0");
18    }
19
20    m_grossSales = grossSales;
21 }
22
23 // return gross sales amount
24 double CommissionEmployee::getGrossSales() const {return m_grossSales;}
25
26 // set commission rate
27 void CommissionEmployee::setCommissionRate(double commissionRate) {
28    if (commissionRate <= 0.0 || commissionRate >= 1.0) {
29       throw std::invalid_argument(
30          "Commission rate must be > 0.0 and < 1.0");
31    }
32
33    m_commissionRate = commissionRate;
34 }
35
36 // return commission rate
37 double CommissionEmployee::getCommissionRate() const {
38    return m_commissionRate;
39 }
40
41 // calculate earnings
42 double CommissionEmployee::getPay() const {
43    return getGrossSales() * getCommissionRate();
44 }
45
46 // return string representation of CommissionEmployee object
47 std::string CommissionEmployee::getString() const {
48    return fmt::format(
49       "{}\n{}: ${:.2f}\n{}: {:.2f}", Employee::getString(),
50       "gross sales", getGrossSales(),
51       "commission rate", getCommissionRate());
52 }
```

Fig. 10.24 `CommissionEmployee` class member-function definitions.

Runtime Polymorphism with the Employee Hierarchy Using NVI

SE The test application for this example is identical to the one in Fig. 10.17, so we show only the output in Fig. 10.25. The program's output also is identical to Fig. 10.17, demonstrating that we still get polymorphic processing even with `protected` and `private` base-class `virtual` functions. **Our client**

code now calls only non-`virtual` functions. Yet, each derived class provided custom behavior by overriding the `protected` and `private` base-class `virtual` functions. The `virtual` functions are now internal implementation details of the class hierarchy, hidden from the client-code programmer. We can change those `virtual` function implementations—and potentially even their signatures—without affecting the client code.

```
EMPLOYEES PROCESSED INDIVIDUALLY VIA VARIABLE NAMES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

EMPLOYEES PROCESSED POLYMORPHICALLY VIA DYNAMIC BINDING

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS POINTERS
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

VIRTUAL FUNCTION CALLS MADE VIA BASE-CLASS REFERENCES
name: John Smith
salary: $800.00
earned $800.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00
```

Fig. 10.25 Processing `Employee` derived-class objects with static binding, then polymorphically using dynamic binding.

10.12 Program to an Interface, Not an Implementation[24]

Implementation inheritance is primarily used to define closely related classes with many of the same data members and member function implementations. This kind of inheritance creates **tightly coupled** classes in which the base class's data members and member functions are inherited into derived classes. Changes to a base class directly affect all corresponding derived classes.

24. Defined in Gamma et al., pp. 17–18; also discussed in Joshua Bloch, *Effective Java*. Upper Saddle River, NJ: Addison-Wesley, 2008.

SE **Tight coupling can make modifying class hierarchies difficult.** Consider how you might modify Section 10.9's `Employee` hierarchy to support retirement plans. There are many different retirement plan types, including 401(k)s and IRAs. We might add a pure `virtual makeRetirementDeposit` member function to the class `Employee`. Then we'd define various derived classes such as `SalariedEmployeeWith401k`, `SalariedEmployee-WithIRA`, `CommissionEmployeeWith401k`, `CommissionEmployeeWithIRA`, etc., each with an appropriate `makeRetirementDeposit` implementation. As you can see, you quickly wind up with a proliferation of derived classes, making the hierarchy more challenging to implement and maintain.

Small inheritance hierarchies under the control of one person tend to be more manageable than large ones maintained by many people. This is true even with the tight coupling associated with implementation inheritance.

Rethinking the `Employee` Hierarchy: Composition and Dependency Injection

Over the years, programmers have written numerous papers, articles and blog posts about the problems with tightly coupled class hierarchies like Section 10.9's `Employee` hierarchy. In this example, we'll refactor our `Employee` hierarchy so that the `Employee`'s compensation model is not "hardwired" into the class hierarchy.[25]

25. We'd like to thank Brian Goetz, Oracle's Java Language Architect, for suggesting the class architecture we use in this section when he reviewed a recent edition of our book *Java How to Program*.

SE **To do so, we'll use composition and dependency injection in which a class contains a pointer to an object that provides a behavior required by objects of the class.** In our `Employee` payroll example, that behavior is calculating each `Employee`'s earnings. We'll define a new `Employee` class that *has a* pointer to a `CompensationModel` object with `earnings` and `toString` member functions. This `Employee` class will not be a base class—to emphasize that, we'll make it `final`. We'll then define derived classes of class `CompensationModel` that implement how `Employee`s get compensated:

- fixed salary and
- commission based on gross sales.

Of course, we could define other `CompensationModel`s, too.

Interface Inheritance Is Best for Flexibility

CG For our `CompensationModel`s, we'll use **interface inheritance**. Each `CompensationModel` concrete class will inherit from a class containing **only pure `virtual` functions**. Such a class is called an **interface** or a **pure abstract class**. The C++ Core Guidelines recommend inheriting from pure abstract classes rather than classes with implementation details.[26,27,28,29]

26. "I.25: Prefer Abstract Classes as Interfaces to Class Hierarchies." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ri-abstract.
27. "C.121: If a Base Class Is Used as an Interface, Make It a Pure Abstract Class." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-abstract.
28. "C.122: Use Abstract Classes As Interfaces When Complete Separation of Interface and Implementation Is Needed." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-separation.

29. “C.129: When Designing a Class Hierarchy, Distinguish Between Implementation Inheritance and Interface Inheritance.” Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-kind.

SE **Interfaces typically do not have data members.** Interface inheritance may require more work than implementation inheritance because concrete classes must provide both data and implementations of the interface’s pure `virtual` member functions, even if they’re similar or identical among classes. As we’ll discuss at the end of the example, this approach gives you additional flexibility by eliminating the tight coupling between classes. The discussion of device drivers in the context of abstract classes at the end of Section 10.8 is a good example of how interfaces enable systems to be modified easily.

10.12.1 Rethinking the Employee Hierarchy—`CompensationModel` Interface

Let’s reconsider Section 10.9’s `Employee` hierarchy with composition and an interface. We can say that each `Employee` *has a* `CompensationModel`. Figure 10.26 defines the **interface `CompensationModel`**, which is a pure abstract class, so it does not have a `.cpp` file. The class has a compiler-generated `virtual` destructor and two pure `virtual` functions:

- `earnings` to calculate an employee’s pay, based on its `CompensationModel`, and
- `toString` to create a string representation of a `CompensationModel`.

Any class that inherits from `CompensationModel` and overrides its pure `virtual` functions *is a* `CompensationModel` that implements this interface.

```
1  // Fig. 10.26: CompensationModel.h
2  // CompensationModel "interface" is a pure abstract base class.
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5
6  class CompensationModel {
7  public:
8     virtual ~CompensationModel() = default; // generated destructor
9     virtual double earnings() const = 0; // pure virtual
10    virtual std::string toString() const = 0; // pure virtual
11 };
```

Fig. 10.26 `CompensationModel` “interface” is a pure abstract base class.

10.12.2 Class `Employee`

Figure 10.27 defines our new `Employee` class. **Each `Employee` has a pointer to the implementation of its `CompensationModel`** (line 16). Note that this class is declared `final` so that it cannot be used as a base class.

```
1  // Fig. 10.27: Employee.h
2  // An Employee "has a" CompensationModel.
3  #pragma once // prevent multiple inclusions of header
4  #include <string>
5  #include <string_view>
6  #include "CompensationModel.h"
7
```

```
 8   class Employee final {
 9   public:
10      Employee(std::string_view name, CompensationModel* modelPtr);
11      void setCompensationModel(CompensationModel *modelPtr);
12      double earnings() const;
13      std::string toString() const;
14   private:
15      std::string m_name{};
16      CompensationModel* m_modelPtr{}; // pointer to an implementation object
17   };
```

Fig. 10.27 An `Employee` "has a" `CompensationModel`.

Figure 10.28 defines class `Employee`'s member functions:

- The constructor (lines 10–11) initializes the `Employee`'s name and aims its `CompensationModel` pointer at an object that implements interface `Compensation-Model`. This technique is known as **constructor injection**. The constructor receives a pointer (or reference) to another object and stores it in the object being constructed.[30]

Err 30. With dependency injection, the `CompensationModel` object must exist longer than the `Employee` object to prevent a runtime logic error known as a dangling pointer. We discuss dangling pointers in Section 11.6.

- The `setCompensationModel` member function (lines 15–17) enables the client code to change an `Employee`'s `CompensationModel` by aiming `m_modelPtr` at a different `CompensationModel` object. This technique is known as **property injection**.
- The `earnings` member function (lines 20–22) determines the `Employee`'s earnings by calling the `CompensationModel` implementation's `earnings` member function via the `CompensationModel` pointer `m_modelPtr`.
- The `toString` member function (lines 25–27) creates an `Employee`'s `string` representation consisting of the `Employee`'s name, followed by the compensation information, which we get by calling the `CompensationModel` implementation's `toString` member function via the `CompensationModel` pointer `m_modelPtr`.

SE Constructor injection and property injection are both forms of **dependency injection**. **You specify part of an object's behavior by providing it with a pointer or reference to an object that defines the behavior.**[31] In this example, a `CompensationModel` provides the behavior that enables an `Employee` to calculate its `earnings` and to generate a `string` representation.

31. "Dependency Injection." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Dependency_injection.

```
 1   // Fig. 10.28: Employee.cpp
 2   // Class Employee member-function definitions.
 3   #include <fmt/format.h>
 4   #include <string>
 5   #include "CompensationModel.h"
 6   #include "Employee.h"
 7
 8   // constructor performs "constructor injection" to initialize
 9   // the CompensationModel pointer to a CompensationModel implementation
10   Employee::Employee(std::string_view name, CompensationModel* modelPtr)
11      : m_name{name}, m_modelPtr{modelPtr} {}
12
13   // set function performs "property injection" to change the
```

```
14   // CompensationModel pointer to a new CompensationModel implementation
15   void Employee::setCompensationModel(CompensationModel* modelPtr) {
16      m_modelPtr = modelPtr;
17   }
18
19   // use the CompensationModel to calculate the Employee's earnings
20   double Employee::earnings() const {
21      return m_modelPtr->earnings();
22   };
23
24   // return string representation of Employee object
25   std::string Employee::toString() const {
26      return fmt::format("{}\n{}", m_name, m_modelPtr->toString());
27   }
```

Fig. 10.28 Class `Employee` member-function definitions.

10.12.3 `CompensationModel` Implementations

Next, let's define our `CompensationModel` implementations. Objects of these classes will be injected into `Employee` objects to specify how to calculate their earnings.

`Salaried` Derived Class of `CompensationModel`

A `Salaried` compensation model (Figs. 10.29–10.30) defines how to pay an `Employee` who receives a fixed salary. The class contains an `m_salary` data member and overrides interface `CompensationModel`'s `earnings` and `toString` member functions. **`Salaried` is declared `final` (line 7), so it is a leaf node in the `CompensationModel` hierarchy. It may not be used as a base class.**

```
1   // Fig. 10.29: Salaried.h
2   // Salaried implements the CompensationModel interface.
3   #pragma once
4   #include <string>
5   #include "CompensationModel.h" // CompensationModel definition
6
7   class Salaried final : public CompensationModel {
8   public:
9      explicit Salaried(double salary);
10     double earnings() const override;
11     std::string toString() const override;
12  private:
13     double m_salary{0.0};
14  };
```

Fig. 10.29 `Salaried` implements the `CompensationModel` interface.

```
1   // Fig. 10.30: Salaried.cpp
2   // Salaried compensation model member-function definitions.
3   #include <fmt/format.h>
4   #include <stdexcept>
5   #include "Salaried.h" // class definition
6
7   // constructor
8   Salaried::Salaried(double salary) : m_salary{salary} {
```

```
9       if (m_salary < 0.0) {
10         throw std::invalid_argument("Weekly salary must be >= 0.0");
11      }
12   }
13
14   // override CompensationModel pure virtual function earnings
15   double Salaried::earnings() const {return m_salary;}
16
17   // override CompensationModel pure virtual function toString
18   std::string Salaried::toString() const {
19      return fmt::format("salary: ${:.2f}", m_salary);
20   }
```

Fig. 10.30 `Salaried` compensation model member-function definitions.

`Commission` Derived Class of CompensationModel

A `Commission` compensation model (Figs. 10.31–10.32) defines how to pay an `Employee` commission based on gross sales. The class contains data members `m_grossSales` and `m_commissionRate` and overrides interface `CompensationModel`'s `earnings` and `toString` member functions. Like class `Salaried`, class `Commission` is declared `final` (line 7), so it is a leaf node in the `CompensationModel` hierarchy. It may not be used as a base class.

```
1   // Fig. 10.31: Commission.h
2   // Commission implements the CompensationModel interface.
3   #pragma once
4   #include <string>
5   #include "CompensationModel.h" // CompensationModel definition
6
7   class Commission final : public CompensationModel {
8   public:
9      Commission(double grossSales, double commissionRate);
10     double earnings() const override;
11     std::string toString() const override;
12  private:
13     double m_grossSales{0.0};
14     double m_commissionRate{0.0};
15  };
```

Fig. 10.31 `Commission` implements the `CompensationModel` interface.

```
1   // Fig. 10.32: Commission.cpp
2   // Commission member-function definitions.
3   #include <fmt/format.h>
4   #include <stdexcept>
5   #include "Commission.h" // class definition
6
7   // constructor
8   Commission::Commission(double grossSales, double commissionRate)
9      : m_grossSales{grossSales}, m_commissionRate{commissionRate} {
10
11     if (m_grossSales < 0.0) {
12        throw std::invalid_argument("Gross sales must be >= 0.0");
13     }
```

```
14
15      if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
16         throw std::invalid_argument(
17            "Commission rate must be > 0.0 and < 1.0");
18      }
19   }
20
21   // override CompensationModel pure virtual function earnings
22   double Commission::earnings() const {
23      return m_grossSales * m_commissionRate;
24   }
25
26   // override CompensationModel pure virtual function toString
27   std::string Commission::toString() const {
28      return fmt::format("gross sales: ${:.2f}; commission rate: {:.2f}",
29         m_grossSales, m_commissionRate);
30   }
```

Fig. 10.32 `Commission` member-function definitions.

10.12.4 Testing the New Hierarchy

We've created our `CompensationModel` **interface** and derived-class implementations defining how `Employee`s get paid. Now, let's create `Employee` objects and initialize each with an appropriate concrete `CompensationModel` implementation (Fig. 10.33).

```
1   // fig10_33.cpp
2   // Processing Employees with various CompensationModels.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <vector>
6   #include "Employee.h"
7   #include "Salaried.h"
8   #include "Commission.h"
9
10  int main() {
11     // create CompensationModels and Employees
12     Salaried salaried{800.0};
13     Employee salariedEmployee{"John Smith", &salaried};
14
15     Commission commission{10000, .06};
16     Employee commissionEmployee{"Sue Jones", &commission};
17
18     // create and initialize vector of Employees
19     std::vector employees{salariedEmployee, commissionEmployee};
20
21     // print each Employee's information and earnings
22     for (const Employee& employee : employees) {
23        std::cout << fmt::format("{}\nearned: ${:.2f}\n\n",
24           employee.toString(), employee.earnings());
25     }
26  }
```

```
John Smith
salary: $800.00
```

```
earned: $800.00

Sue Jones
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Fig. 10.33 Processing `Employees` with various `CompensationModels`.

Line 12 creates a `Salaried` compensation model object (`salaried`). Line 13 creates the `Employee` `salariedEmployee` and injects its `CompensationModel`, passing a pointer to `salaried` as the second constructor argument. Line 15 creates a `Commission` compensation model object (`commission`). Line 16 creates the `Employee commissionEmployee` and injects its `CompensationModel`, passing a pointer to `commission` as the second constructor argument.[32] Line 19 creates a `vector` of `Employees` and initializes it with the `salariedEmployee` and `commissionEmployee` objects. Finally, lines 22–25 iterate through the `vector`, displaying each `Employee`'s string representation and earnings.

32. In this example, both `CompensationModels` will outlive their corresponding `Employee` objects, so this example does not have the potential for the runtime logic error mentioned in footnote 30.

10.12.5 Dependency Injection Design Benefits

Flexibility If `CompensationModels` Change

Declaring the `CompensationModels` as separate classes that implement the same interface provides flexibility for future changes. Suppose a company adds new ways to pay employees. We can simply define a new `CompensationModel` derived class with an appropriate `earnings` function.

Flexibility If `Employees` Are Promoted

SE The interface-based, composition-and-dependency-injection approach used in this example is more flexible than Section 10.9's class hierarchy. In Section 10.9, if an `Employee` were promoted, you'd need to change its object type by creating a new object of the appropriate `Employee` derived class, then moving data into the new object. **Using dependency injection, you can simply call `Employee`'s `setCompensationModel` member function and inject a pointer to a different `CompensationModel` that replaces the existing one.**

Flexibility If `Employees` Acquire New Capabilities

The interface-based composition and dependency-injection approach is also more flexible for enhancing class **`Employee`.** If we decide to support retirement plans (such as 401(k)s and IRAs), we could say that every `Employee` *has a* `RetirementPlan`. First, we'd define interface `RetirementPlan` with a `makeRetirementDeposit` member function and provide appropriate derived-class implementations.

Using interface-based composition and dependency injection, as shown in this example, requires only small changes to class `Employee` to support `RetirementPlans`:

- a data member that points to a `RetirementPlan`,
- one more constructor argument to initialize the `RetirementPlan` pointer and
- a `setRetirementPlan` member function we can call if we ever need to change the `RetirementPlan`.

10.13 Runtime Polymorphism with `std::variant` and `std::visit`

17 SE So far, we've achieved runtime polymorphism via implementation inheritance or interface inheritance. As you've seen, both techniques require class hierarchies. What if you have objects of *unrelated* classes, but you'd still like to process those objects polymorphically at runtime?

You can achieve this with C++17's **class template `std::variant`** and the **standard-library function `std::visit`** (both in header `<variant>`).[33,34,35,36] **The caveat is that you must know in advance all the types your program needs to process via runtime polymorphism—known as a closed set of types.** A `std::variant` object can store one object at a time of any type specified when you create the `std::variant` object. As you'll see, you call functions on the objects in a `std::variant` object using the `std::visit` function.

33. Nevin Liber, "The Many Variants of `std::variant`," YouTube Video, June 16, 2019. Accessed January 12, 2020. https://www.youtube.com/watch?v=JUxhwf7gYLg.
34. "`std::variant`." Accessed January 11, 2022. https://en.cppreference.com/w/cpp/utility/variant.
35. Bartlomiej Filipek, "Runtime Polymorphism with `std::variant` and `std::visit`," November 2, 2020. Accessed January 11, 2022. https://www.bfilipek.com/2020/04/variant-virtual-polymorphism.html.
36. Bartlomiej Filipek, "Everything You Need to Know About `std::variant` from C++17," June 4, 2018. Accessed January 11, 2022. https://www.bfilipek.com/2018/06/variant.html.

To demonstrate runtime polymorphism with `std::variant`, we'll reimplement our Section 10.12 example. The classes used here are nearly identical, so we'll point out only the differences.

Compensation Model `Salaried`

Class `Salaried` (Figs. 10.34–10.35) defines the compensation model for an `Employee` who gets paid a fixed salary. The only difference between this class and the one in Section 10.12 is that **this `Salaried` is not a derived class. So, its `earnings` and `toString` member functions do not override base-class `virtual` functions**.

```
1  // Fig. 10.34: Salaried.h
2  // Salaried compensation model.
3  #pragma once
4  #include <string>
5
6  class Salaried {
7  public:
8     Salaried(double salary);
9     double earnings() const;
10    std::string toString() const;
11 private:
12    double m_salary{0.0};
13 };
```

Fig. 10.34 Salaried compensation model.

```
1  // Fig. 10.35: Salaried.cpp
2  // Salaried compensation model member-function definitions.
3  #include <fmt/format.h>
4  #include <stdexcept>
5  #include "Salaried.h" // class definition
6
7  // constructor
8  Salaried::Salaried(double salary) : m_salary{salary} {
9     if (m_salary < 0.0) {
10       throw std::invalid_argument("Weekly salary must be >= 0.0");
11    }
```

```
   }

   // calculate earnings
   double Salaried::earnings() const {return m_salary;}

   // return string containing Salaried compensation model information
   std::string Salaried::toString() const {
      return fmt::format("salary: ${:.2f}", m_salary);
   }
```

Fig. 10.35 Salaried compensation model member-function definitions.

Compensation Model `Commission`

Class `Commission` (Figs. 10.36–10.37) defines the compensation model for an `Employee` who gets paid commission based on gross sales. Like `Salaried`, this `Commission` is *not* a derived class. So, its `earnings` and `toString` member functions do not override base-class `virtual` functions, as they did in Section 10.12.

```
// Fig. 10.36: Commission.h
// Commission compensation model.
#pragma once
#include <string>

class Commission {
public:
   Commission(double grossSales, double commissionRate);
   double earnings() const;
   std::string toString() const;
private:
   double m_grossSales{0.0};
   double m_commissionRate{0.0};
};
```

Fig. 10.36 Commission compensation model.

```
// Fig. 10.37: Commission.cpp
// Commission member-function definitions.
#include <fmt/format.h>
#include <stdexcept>
#include "Commission.h" // class definition

// constructor
Commission::Commission(double grossSales, double commissionRate)
   : m_grossSales{grossSales}, m_commissionRate{commissionRate} {

   if (m_grossSales < 0.0) {
      throw std::invalid_argument("Gross sales must be >= 0.0");
   }

   if (m_commissionRate <= 0.0 || m_commissionRate >= 1.0) {
      throw std::invalid_argument(
         "Commission rate must be > 0.0 and < 1.0");
```

```
18      }
19   }
20
21   // calculate earnings
22   double Commission::earnings() const {
23      return m_grossSales * m_commissionRate;
24   }
25
26   // return string containing Commission information
27   std::string Commission::toString() const {
28      return fmt::format("gross sales: ${:.2f}; commission rate: {:.2f}",
29         m_grossSales, m_commissionRate);
30   }
```

Fig. 10.37 Commission member-function definitions.

`Employee` Class Definition

As in Section 10.12, each `Employee` (Fig. 10.38) has a compensation model (line 21). However, in this example, the compensation model is a `std::variant` object containing either a `Commission` *or* a `Salaried` object—**note that this is an object, not a pointer to an object**. Line 11's `using` declaration:

```
using CompensationModel = std::variant<Commission, Salaried>;
```

```
 1   // Fig. 10.38: Employee.h
 2   // An Employee "has a" CompensationModel.
 3   #pragma once // prevent multiple inclusions of header
 4   #include <string>
 5   #include <string_view>
 6   #include <variant>
 7   #include "Commission.h"
 8   #include "Salaried.h"
 9
10   // define a convenient name for the std::variant type
11   using CompensationModel = std::variant<Commission, Salaried>;
12
13   class Employee {
14   public:
15      Employee(std::string_view name, CompensationModel model);
16      void setCompensationModel(CompensationModel model);
17      double earnings() const;
18      std::string toString() const;
19   private:
20      std::string m_name{};
21      CompensationModel m_model; // note this is not a pointer
22   };
```

Fig. 10.38 An `Employee` "has a" `CompensationModel`.

defines the alias `CompensationModel` for the `std::variant` type:

```
std::variant<Commission, Salaried>
```

11 SE Such `using` declarations (C++11) are known as **alias declarations**. They enable you to create convenient names for complex types—such as a `std::variant` type that potentially could have many type parameters. **At any given time, an object of our `std::variant` type can store either a `Commission` object or a `Salaried` object**. We use our `Compensation-Model` alias in line 21 to define the `std::variant` object that stores the `Employee`'s compensation model.

Type-Safe union

In C and C++, a **`union`**[37] is a region of memory that, over time, can contain objects of a variety of types. A `union`'s members share the same storage space, so a `union` may contain a maximum of one object at a time and requires enough memory to hold the largest of its members. **A `std::variant` object is often referred to as a type-safe `union`.**[38]

37. "Union Declaration." Accessed January 11, 2022. https://en.cppreference.com/w/cpp/language/union.
38. "`std::variant`." Accessed January 11, 2022. https://en.cppreference.com/w/cpp/utility/variant.

`Employee` Constructor and `setCompensationModel` Member Function

Class `Employee`'s member functions (Fig. 10.39) perform the same tasks as in Fig. 10.28 with several modifications. The `Employee` class's constructor (lines 8–9) and `setCompensationModel` member function (lines 12–14) each receive a **`CompensationModel` object**. Unlike the composition-and-dependency-injection approach, the `std::variant` object **stores an actual object**, not a pointer to one.

```
1  // Fig. 10.39: Employee.cpp
2  // Class Employee member-function definitions.
3  #include <fmt/format.h>
4  #include <string>
5  #include "Employee.h"
6
7  // constructor
8  Employee::Employee(std::string_view name, CompensationModel model)
9     : m_name{name}, m_model{model} {}
10
11 // change the Employee's CompensationModel
12 void Employee::setCompensationModel(CompensationModel model) {
13    m_model = model;
14 }
15
16 // return the Employee's earnings
17 double Employee::earnings() const {
18    auto getEarnings{[](const auto& model){return model.earnings();}};
19    return std::visit(getEarnings, m_model);
20 }
21
22 // return string representation of an Employee object
23 std::string Employee::toString() const {
24    auto getString{[](const auto& model){return model.toString();}};
25    return fmt::format("{}\n{}", m_name, std::visit(getString, m_model));
26 }
```

Fig. 10.39 Class `Employee` member-function definitions.

Employee `earnings` and `toString` Member Functions: Calling Member Functions with `std::visit`

A key difference between runtime polymorphism via class hierarchies and runtime polymorphism via `std::variant` is that **a `std::variant` object cannot call member functions of the object it contains. Instead, you use the standard library function `std::visit` to invoke a function on the object stored in the `std::variant`.** Consider lines 18–19 in member-function `earnings`:

```
auto getEarnings{[](const auto& model){return model.earnings();}};
return std::visit(getEarnings, m_model);
```

Line 18 defines the variable `getEarnings` and initializes it with a generic lambda expression that receives a reference to an object (`model`) and calls the object's `earnings` member function. This lambda expression can call `earnings` on *any object* with an `earnings` member function that takes no arguments and returns a value. Line 19 passes to function `std::visit` the `getEarnings` lambda expression and the `std::variant` object `m_model`. Function `std::visit` passes `m_model` to the lambda expression, then returns the result of calling `m_model`'s `earnings` member function.

Similarly, line 24 in `Employee`'s `toString` member function creates a lambda expression that returns the result of calling some object's `toString` member function. Line 25 calls `std::visit` to pass `m_model` to the lambda expression, which returns `m_model`'s `toString` result.

Testing Runtime Polymorphism with `std::variant` and `std::visit`

In `main`, lines 11–12 create two `Employee` objects—the first stores a `Salaried` object in its `std::variant` and the second stores a `Commission` object. The expression

```
Salaried{800.0}
```

in line 11 creates a `Salaried` object, which is immediately used to initialize the `Employee`'s `CompansationModel`. Similarly, the expression

```
Commission{10000.0, .06}
```

in line 12 creates a `Commission` object, which is immediately used to initialize the `Employee`'s `CompansationModel`.

```
 1   // fig10_40.cpp
 2   // Processing Employees with various compensation models.
 3   #include <fmt/format.h>
 4   #include <iostream>
 5   #include <vector>
 6   #include "Employee.h"
 7   #include "Salaried.h"
 8   #include "Commission.h"
 9
10   int main() {
11      Employee salariedEmployee{"John Smith", Salaried{800.0}};
12      Employee commissionEmployee{"Sue Jones", Commission{10000.0, .06}};
13
14      // create and initialize vector of three Employees
15      std::vector employees{salariedEmployee, commissionEmployee};
16
```

```
17      // print each Employee's information and earnings
18      for (const Employee& employee : employees) {
19         std::cout << fmt::format("{}\nearned: ${:.2f}\n\n",
20            employee.toString(), employee.earnings());
21      }
22   }
```

```
John Smith
salary: $800.00
earned: $800.00

Sue Jones
gross sales: $10000.00; commission rate: 0.06
earned: $600.00
```

Fig. 10.40 Processing `Employees` with various compensation models.

SE Line 15 creates a `vector` of `Employee`s, then lines 18–21 iterate through it, calling each `Employee`'s `earnings` and `toString` member functions to demonstrate the runtime polymorphic processing, producing the same results as in Section 10.12. **This ability to invoke common functionality on objects whose types are not related by a class hierarchy is often called duck typing**: *"If it walks like a duck and it quacks like a duck, then it must be a duck."*[39]

39. "Duck Typing." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Duck_typing.

That is, if an object has the appropriate member functions and its type is specified as a member of the `std::variant`, the object will work in this code.

10.14 Multiple Inheritance

SE So far, we've discussed single inheritance, in which each class is derived from exactly one base class. C++ also supports **multiple inheritance**—a class may inherit the members of two or more base classes. **Multiple inheritance is a complicated feature that should be used only by experienced programmers.** Some multiple-inheritance problems are so subtle that newer programming languages, such as Java and C#, support only single inheritance.[40] **Great care is required to design a system to use multiple inheritance properly. It should not be used when single inheritance and/or composition will do the job.**

40. More precisely, Java and C# support only single *implementation* inheritance. They do allow multiple interface inheritance.

SE A common problem with multiple inheritance is that each base class might contain data members or member functions with the same name. This can lead to ambiguity problems when you compile. **The `isocpp.org` FAQ recommends doing multiple inheritance from only pure abstract base classes** to avoid this problem and others we'll discuss in this section and Section 10.14.1.[41]

41. "Inheritance—Multiple and Virtual Inheritance." Accessed January 11, 2022. https://isocpp.org/wiki/faq/multiple-inheritance.

Multiple-Inheritance Example

Let's consider a multiple-inheritance example using implementation inheritance (Figs. 10.41–10.45). Class `Base1` (Fig. 10.41) contains

- one `private int` data member (`m_value`; line 11),
- a constructor (line 8) that sets `m_value` and
- a `public` member function `getData` (line 9) that returns `m_value`.

```
1   // Fig. 10.41: Base1.h
2   // Definition of class Base1
3   #pragma once
4
5   // class Base1 definition
6   class Base1 {
7   public:
8      explicit Base1(int value) : m_value{value} {}
9      int getData() const {return m_value;}
10  private: // accessible to derived classes via getData member function
11     int m_value;
12  };
```

Fig. 10.41 Demonstrating multiple inheritance—`Base1.h`.

Class `Base2` (Fig. 10.42) is similar to class `Base1`, except its `private` data is a `char` named `m_letter` (line 11). Like class `Base1`, `Base2` has a `public` member function `get-Data`, but this function returns `m_letter`'s value.

```
1   // Fig. 10.42: Base2.h
2   // Definition of class Base2
3   #pragma once
4
5   // class Base2 definition
6   class Base2 {
7   public:
8      explicit Base2(char letter) : m_letter{letter} {}
9      char getData() const {return m_letter;}
10  private: // accessible to derived classes via getData member function
11     char m_letter;
12  };
```

Fig. 10.42 Demonstrating multiple inheritance—`Base2.h`.

Class `Derived` (Figs. 10.43–10.44) inherits from classes `Base1` and `Base2` via multiple inheritance. Class `Derived` has

- a `private` data member of type `double` named `m_real` (Fig. 10.43, line 18),
- a constructor to initialize all the data of class `Derived`,
- a `public` member function `getReal` that returns the value of `m_real` and
- a `public` member function `toString` that returns a `string` representation of a `Derived` object.

```
1   // Fig. 10.43: Derived.h
2   // Definition of class Derived which inherits
3   // multiple base classes (Base1 and Base2).
4   #pragma once
5   #include <iostream>
```

```
 6   #include <string>
 7   #include "Base1.h"
 8   #include "Base2.h"
 9   using namespace std;
10
11   // class Derived definition
12   class Derived : public Base1, public Base2 {
13   public:
14      Derived(int value, char letter, double real);
15      double getReal() const;
16      std::string toString() const;
17   private:
18      double m_real; // derived class's private data
19   };
```

Fig. 10.43 Demonstrating multiple inheritance—`Derived.h`.

```
 1   // Fig. 10.44: Derived.cpp
 2   // Member-function definitions for class Derived
 3   #include <fmt/format.h> // In C++20, this will be #include <format>
 4   #include "Derived.h"
 5
 6   // constructor for Derived calls Base1 and Base2 constructors
 7   Derived::Derived(int value, char letter, double real)
 8      : Base1{value}, Base2{letter}, m_real{real} {}
9
10   // return real
11   double Derived::getReal() const {return m_real;}
12
13   // display all data members of Derived
14   std::string Derived::toString() const {
15      return fmt::format("int: {}; char: {}; double: {}",
16         Base1::getData(), Base2::getData(), getReal());
17   }
```

Fig. 10.44 Demonstrating multiple inheritance—`Derived.cpp`.

For multiple inheritance (in Fig. 10.43), we follow the colon (`:`) after `class Derived` with a **comma-separated list of base classes** (line 13). In Fig. 10.44, notice that constructor `Derived` explicitly calls base-class constructors for each base class—`Base1` and `Base2`—using the member-initializer syntax (line 8). **The base-class constructors are called in the order that the inheritance is specified**. If the member-initializer list does not explicitly call a base class's constructor, the base class's default constructor will be called implicitly.

Resolving Ambiguity Issues That Arise When a Derived Class Inherits Member Functions of the Same Name from Multiple Base Classes

Member function `toString` (lines 14–17) returns a `string` representation of a `Derived` object's contents. It uses all of the `Derived` class's *get* member functions. However, there's an ambiguity problem. A **`Derived` object contains two `getData` functions**—one inherited from class `Base1` and one inherited from class `Base2`. This problem is easy to solve by using the scope-resolution operator. `Base1::getData()` gets the value of the variable inherited from class `Base1` (i.e., the `int` variable named `m_value`), and `Base2::getData()` gets the value of the variable inherited from class `Base2` (i.e., the `char` variable named `m_letter`).

Testing the Multiple-Inheritance Hierarchy

Figure 10.45 tests the classes in Figs. 10.41–10.44. Line 10 creates `Base1` object `base1` and initializes it to the `int` value `10`. Line 11 creates `Base2` object `base2` and initializes it to the `char` value `'Z'`. Line 12 creates `Derived` object `derived` and initializes it to contain the `int` value `7`, the `char` value `'A'` and the `double` value `3.5`.

```
1   // fig10_45.cpp
2   // Driver for multiple-inheritance example.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include "Base1.h"
6   #include "Base2.h"
7   #include "Derived.h"
8
9   int main() {
10     Base1 base1{10}; // create Base1 object
11     Base2 base2{'Z'}; // create Base2 object
12     Derived derived{7, 'A', 3.5}; // create Derived object
13
14     // print data in each object
15     std::cout << fmt::format("{}: {}\n{}: {}\n{}: {}\n\n",
16        "Object base1 contains", base1.getData(),
17        "Object base2 contains the character", base2.getData(),
18        "Object derived contains", derived.toString());
19
20     // print data members of derived-class object
21     // scope resolution operator resolves getData ambiguity
22     std::cout << fmt::format("{}\n{}: {}\n{}: {}\n{}: {}\n\n",
23        "Data members of Derived can be accessed individually:",
24        "int", derived.Base1::getData(),
25        "char", derived.Base2::getData(),
26        "double", derived.getReal());
27
28     std::cout << "Derived can be treated as an object"
29        << " of either base class:\n";
30
31     // treat Derived as a Base1 object
32     Base1* base1Ptr = &derived;
33     std::cout << fmt::format("base1Ptr->getData() yields {}\n",
34        base1Ptr->getData());
35
36      // treat Derived as a Base2 object
37     Base2* base2Ptr = &derived;
38     std::cout << fmt::format("base2Ptr->getData() yields {}\n",
39        base2Ptr->getData());
40  }
```

```
Object base1 contains: 10
Object base2 contains the character: Z
Object derived contains: int: 7; char: A; double: 3.5

Data members of Derived can be accessed individually:
int: 7
char: A
```

```
double: 3.5

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A
```

Fig. 10.45 Demonstrating multiple inheritance.

Lines 15–18 display each object's data values. For objects `base1` and `base2`, we invoke each object's `getData` member function. Even though there are *two* `getData` functions in this example, the calls are not ambiguous. In line 16, the compiler knows that `base1` is an object of class `Base1`, so class `Base1`'s `getData` is called. In line 17, the compiler knows that `base2` is an object of class `Base2`, so class `Base2`'s `getData` is called. Line 18 gets `derived`'s contents by calling its `toString` member function.

Lines 22–26 output `derived`'s contents again by using class `Derived`'s *get* member functions. Again, there is an *ambiguity* problem—this object contains `getData` functions from both class `Base1` and class `Base2`. The expression

```
derived.Base1::getData()
```

gets the value of `m_value` inherited from class `Base1` and

```
derived.Base2::getData()
```

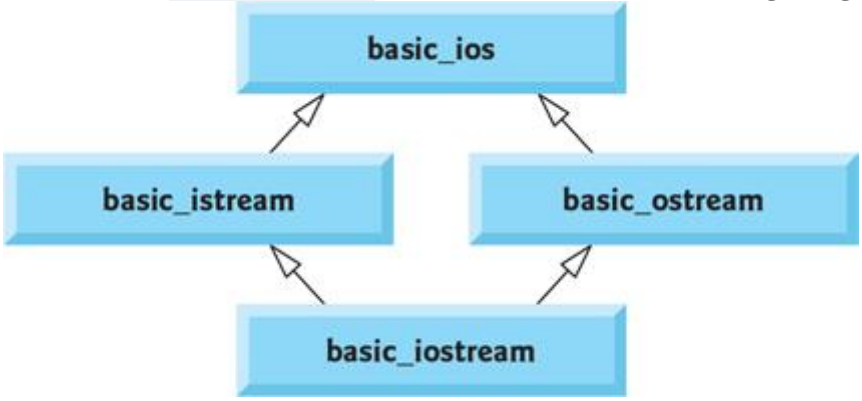
gets the value of `m_letter` inherited from class `Base2`.

Demonstrating the *Is-a* Relationships in Multiple Inheritance

The is-a relationships of single inheritance also apply in multiple-inheritance relationships. To demonstrate this, line 32 assigns `derived`'s address to the `Base1` pointer `base1Ptr`. This is allowed because **a `Derived` object is a `Base1` object**. Line 34 invokes `Base1` member function `getData` via `base1Ptr` to obtain the value of only the `Base1` part of the object `derived`. Line 37 assigns `derived`'s address to the `Base2` pointer `base2Ptr`. This is allowed because **a `Derived` object is a `Base2` object**. Line 39 invokes `Base2` member function `getData` via `base2Ptr` to obtain the value of only the `Base2` part of the object `derived`.

10.14.1 Diamond Inheritance

The preceding example showed *multiple inheritance*, the process by which one class inherits from *two or more* classes. Multiple inheritance is used, for example, in the C++ standard library to form class `basic_iostream`, as shown in the following diagram:



Class `basic_ios` is the base class of both `basic_istream` and `basic_ostream`. Each is formed with *single inheritance*. Class `basic_iostream` inherits from both `basic_istream` and `basic_ostream`. This enables class `basic_iostream` objects to provide the functionality of `basic_istream`s and `basic_ostream`s. In multiple-inheritance hierarchies, the inheritance described in this diagram is referred to as **diamond inheritance**.

Classes `basic_istream` and `basic_ostream` each inherit from `basic_ios`, so a potential problem exists for `basic_iostream`. It could inherit *two* copies of `basic_ios`'s members—one via `basic_istream` and one via `basic_ostream`. This would be *ambiguous* and result in a compilation error—the compiler would not know which copy of `basic_ios`'s members to use. Let's see how **using `virtual` base classes solves this problem.**

Compilation Errors Produced When Ambiguity Arises in Diamond Inheritance

Figure 10.46 demonstrates the *ambiguity* that can occur in *diamond inheritance*. Class `Base` (lines 7–10) contains pure `virtual` function `print` (line 9). Classes `DerivedOne` (lines 13–17) and `DerivedTwo` (lines 20–24) each publicly inherit from `Base` and override function `print`. Class `DerivedOne` and class `DerivedTwo` each contain a **base-class subobject**—the members of class `Base` in this example.

```
1  // fig10_46.cpp
2  // Attempting to polymorphically call a function that is
3  // inherited from each of two base classes.
4  #include <iostream>
5
6  // class Base definition
7  class Base {
8  public:
9     virtual void print() const = 0; // pure virtual
10 };
11
12 // class DerivedOne definition
13 class DerivedOne : public Base {
14 public:
15    // override print function
16    void print() const override {std::cout << "DerivedOne\n";}
17 };
18
19 // class DerivedTwo definition
20 class DerivedTwo : public Base {
21 public:
22    // override print function
23    void print() const override {std::cout << "DerivedTwo\n";}
24 };
25
26 // class Multiple definition
27 class Multiple : public DerivedOne, public DerivedTwo {
28 public:
29    // qualify which version of function print
30    void print() const override {DerivedTwo::print();}
31 };
32
33 int main() {
34    Multiple both{}; // instantiate a Multiple object
```

```
35      DerivedOne one{}; // instantiate a DerivedOne object
36      DerivedTwo two{}; // instantiate a DerivedTwo object
37      Base* array[3]{}; // create array of base-class pointers
38
39      array[0] = &both; // ERROR--ambiguous
40      array[1] = &one;
41      array[2] = &two;
42
43      // polymorphically invoke print
44      for (int i{0}; i < 3; ++i) {
45         array[i] ->print();
46      }
47   }
```

Microsoft Visual C++ compiler error message:

```
fig10_46.cpp(39,20): error C2594: '=': ambiguous conversions from
'Multiple *' to 'Base *'
```

Fig. 10.46 Attempting to polymorphically call a function that is inherited from each of two base classes.

Class `Multiple` (lines 27–31) inherits from *both* class `DerivedOne` and class `DerivedTwo`. In class `Multiple`, function `print` is overridden to call `DerivedTwo`'s `print` (line 30). Notice that we must *qualify* the `print` call—in this case, we used the class name `DerivedTwo`—to specify which version of `print` to call.

Function `main` (lines 33–47) declares objects of classes `Multiple` (line 34), `Derived-One` (line 35) and `DerivedTwo` (line 36). Line 37 declares a built-in array of `Base*` pointers. Each element is assigned the address of an object (lines 39–41). An error occurs when the address of `both`—an object of class `Multiple`—is assigned to `array[0]`. The object `both` actually contains two `Base` subobjects. The compiler does not know which subobject the pointer `array[0]` should point to, so it generates a compilation error indicating an *ambiguous conversion*.

10.14.2 Eliminating Duplicate Subobjects with `virtual` *Base-Class Inheritance*

The problem of duplicate subobjects is resolved with `virtual` inheritance. Only *one* subobject will appear in the derived class when a base class is inherited as `virtual`. Figure 10.47 revises the program of Fig. 10.46 to use a `virtual` base class.

```
1   // fig10_47.cpp
2   // Using virtual base classes.
3   #include <iostream>
4
5   // class Base definition
6   class Base {
7   public:
8      virtual void print() const = 0; // pure virtual
9   };
10
11  // class DerivedOne definition
12  class DerivedOne : virtual public Base {
13  public:
14     // override print function
```

```
15       void print() const override {std::cout << "DerivedOne\n";}
16    };
17
18    // class DerivedTwo definition
19    class DerivedTwo : virtual public Base {
20    public:
21       // override print function
22       void print() const override {std::cout << "DerivedTwo\n";}
23    };
24
25    // class Multiple definition
26    class Multiple : public DerivedOne, public DerivedTwo {
27    public:
28       // qualify which version of function print
29       void print() const override {DerivedTwo::print();}
30    };
31
32    int main() {
33       Multiple both; // instantiate Multiple object
34       DerivedOne one; // instantiate DerivedOne object
35       DerivedTwo two; // instantiate DerivedTwo object
36       Base* array[3];
37
38       array[0] = &both; // allowed now
39       array[1] = &one;
40       array[2] = &two;
41
42       // polymorphically invoke function print
43       for (int i = 0; i < 3; ++i) {
44          array[i]->print();
45       }
46    }
```

```
DerivedTwo
DerivedOne
DerivedTwo
```

Fig. 10.47 Using `virtual` base classes.

The key change is that classes `DerivedOne` (line 12) and `DerivedTwo` (line 19) each inherit from `Base` using `virtual public Base`. Since both classes inherit from `Base`, they each contain a `Base` subobject. The benefit of `virtual` inheritance is not apparent until class `Multiple` inherits from `DerivedOne` and `DerivedTwo` (line 26). **Since each base class used `virtual` inheritance, the compiler ensures that `Multiple` inherits only one `Base` subobject.** This eliminates the ambiguity error generated by the compiler in Fig. 10.46. The compiler now allows the implicit conversion of the derived-class pointer (`&both`) to the base-class pointer `array[0]` in line 38 in `main`. The `for` statement in lines 43–45 polymorphically calls `print` for each object.

Constructors in Multiple-Inheritance Hierarchies with `virtual` Base Classes

SE Implementing hierarchies with `virtual` base classes is simpler if *default constructors* are used for the base classes. Figures 10.46 and 10.47 use compiler-generated *default constructors*. If a `virtual` base class provides a constructor that requires arguments, the derived-class implementations become more complicated. **The most derived class must explicitly invoke the `virtual` base class's constructor.** For this reason, consider providing a default constructor

for `virtual` base classes, so the derived classes do not need to explicitly invoke the `virtual` base class's constructor.[42,43]

42. "Inheritance—Multiple and Virtual Inheritance : What special considerations do I need to know about when I use virtual inheritance?" Accessed January 11, 2022. https://isocpp.org/wiki/faq/multiple-inheritance#virtual-inheritance-abcs.

43. "Inheritance—Multiple and Virtual Inheritanc.

10.15 `protected` Class Members: A Deeper Look

Chapter 9 introduced the access specifiers `public` and `private`. A base class's `public` members are accessible within its class, and anywhere the program has access to an object of that class or one of its derived classes. A base class's `private` members are accessible only within its body and to its `friend`s. The access specifier **`protected`** offers an intermediate level of protection between `public` and `private` access. Such members are accessible within that base class, by members and `friend`s of that base class, and by members and `friend`s of any classes derived from that base class.

In public inheritance, all `public` and `protected` base-class members retain their original access when they become members of the derived class:

- `public` base-class members become `public` derived-class members and
- `protected` base-class members become `protected` derived-class members.

A base class's `private` members are not accessible outside the class itself. Derived classes can access a base class's `private` members only through the `public` or `protected` member functions inherited from the base class. Derived-class member functions can refer to `public` and `protected` members inherited from the base class by their member names.

Problems with `protected` Data

SE It's best to avoid `protected` data members because they create some serious problems:

- **A derived-class object does not have to use a member function to set a base-class `protected` data member's value. So, an invalid value can be assigned, leaving the object in an inconsistent state.** For example, if `CommissionEmployee`'s data member `m_grossSales` is `protected` (and the class is not `final`), a derived-class object can assign a negative value to `m_grossSales`.
- CG **Derived-class member functions are more likely to be written to depend on the base class's data.** Derived classes should depend only on the base-class non-`private` member functions. If we were to change the name of a base-class `protected` data member, we'd need to modify every derived class that references the data directly. Such software is said to be fragile or brittle. **A small change in the base class can "break" the derived class. This is known as the fragile base-class problem[44] and is a key reason why the C++ Core Guidelines recommend avoiding `protected` data.**[45] You should be able to make changes in a base class without having to modify its derived classes.

44. "Fragile Base Class." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Fragile_base_class.

45. C++ Core Guidelines, "C.133: Avoid `protected` Data." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-protected.

SE In most cases, it's better to use `private` data members to encourage proper software engineering. **Declaring base-class data members `private` allows you to change the base-class implementation without having to change derived-class implementations. This makes your code easier to maintain, modify and debug.**

SE A derived class can change the state of `private` base-class data members only through non-`private` base-class member functions inherited into the derived class. **If a derived class could access its base class's `private` data members, classes that inherit from that derived class could access the data members as well**, and the benefits of information hiding would be lost.

`protected` Base-Class Member Functions

SE One use of `protected` is to **define base-class member functions that should not be exposed to client code but should be accessible in derived classes**. A use-case is to enable a derived class to override the base-class `protected virtual` function and call the original base-class version from the derived class's implementation.[46] We used this capability in the example of Section 10.11, Non-Virtual Interface (NVI) Idiom.

46. Herb Sutter, "Virtuality," September 2001. Accessed January 11, 2022. http://www.gotw.ca/publications/mill18.htm.

10.16 `public`, `protected` and `private` Inheritance

You can choose between **`public`**, **`protected`** or **`private` inheritance**. `public` inheritance is the most common, and `protected` inheritance is rare. The following diagram summarizes the accessibility of base-class members in a derived class for each inheritance type. The first column contains the base-class member access specifiers.

Base-class access specifier	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by me and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by me and `friend` functions.
`private`	Inaccessible in the derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions	Inaccessible in the derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions	Inaccessible in the derived cla Can be accessed by member fu through `public` or `protected` me the base class.

Base-class access specifier	`public` inheritance	`protected` inheritance	`private` inheritance
	inherited from the base class.	inherited from the base class.	

With `public` inheritance

- `public` base-class members become `public` derived-class members and
- `protected` base-class members become `protected` derived-class members.

A base class's `private` members are *never* accessible directly in the derived class but can be accessed by calling inherited `public` and `protected` base-class member functions designed to access those `private` members.

When deriving a class with `protected` inheritance, `public` and `protected` base-class members become `protected` derived-class members. When deriving a class with `private` inheritance, `public` and `protected` base-class members become `private` derived-class members.

SE **Classes created with `private` and `protected` inheritance do not have *is-a* relation-ships with their base classes because the base class's `public` members are not accessible to the derived class's client code.**

Default Inheritance in `class`es vs. `struct`s

Both `class` and `struct` (defined in Section 9.21) can define new types. However, two key differences exist between a `class` and a `struct`. The first is that, by default, `class` members are `private` and `struct` members are `public`. The other difference is in the default inheritance type. A `class` definition like

```
class Derived : Base {
   // ...
};
```

uses `private` inheritance by default, whereas a `struct` definition like

```
struct Derived : Base {
   // ...
};
```

uses `public` inheritance by default.

10.17 More Runtime Polymorphism Techniques; Compile-Time Polymorphism

In this chapter, we focused on inheritance and various runtime polymorphism techniques. This section briefly mentions other techniques for runtime polymorphism and **compile-time polymorphism** (also called **static polymorphism**).[47,48] The idioms and methodologies we discuss here are for developers who want to pursue more advanced techniques. We'll cover some of these in later chapters. For more information on this section's topics, see the corresponding footnotes.

47. "C++: Static Polymorphism." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/C%2B%2B#Static_polymorphism.

48. Kateryna Bondarenko, “Static Polymorphism in C++,” May 6, 2019. Accessed January 11, 2022. https://medium.com/@kateolenya/static-polymorphism-in-c-9e1ae27a945b.

10.17.1 Other Runtime Polymorphism Techniques

You’ve now used inheritance and `virtual` functions to implement runtime polymorphism. You’ve also used the standard-library class template `std::variant` and the `std::visit` function to implement runtime polymorphism for a set of types you know in advance and *without* the need for inheritance and `virtual` functions. Here, we mention other runtime polymorphism techniques.

Runtime Concept Idiom

The **runtime concept idiom**[49,50] was created by Sean Parent.[51] The idiom separates a system’s runtime polymorphic-processing logic from the types it processes polymorphically. Those types do not need to be part of a class hierarchy, so they do not need to override a base-class’s `virtual` functions.

49. Sean Parent, “Inheritance Is the Base Class of Evil,” YouTube Video, September 23, 2013. Accessed January 12, 2022. https://www.youtube.com/watch?v=bIhUE5uUFOA.
50. Sean Parent, “Better Code: Runtime Polymorphism,” YouTube Video, February 27, 2017. Accessed January 12, 2022. https://www.youtube.com/watch?v=QGcVXgEVMJgNDC.
51. “Sean Parent.” Accessed January 13, 2022. https://sean-parent.stlab.cc/.

Consider Section 10.5’s polymorphic video game discussion. The game’s screen manager repeatedly clears and draws the game’s on-screen elements. For this game, the runtime concept idiom’s “concept”[52] is *draw*—every type must have a `draw` function. However, rather than enforcing this requirement via a base class containing a `virtual` function `draw`, the runtime concept idiom uses unrelated classes and duck typing. We’d define `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam` as stand-alone classes. In the runtime concept idiom, these classes’ `draw` functions would be implemented as overloaded *non-member* functions. One argument of each function would be a reference to the type of video-game object to draw. This approach eliminates the tight coupling between classes in inheritance hierarchies, which makes those hierarchies difficult to modify.

52. Not to be confused with C++20’s new concepts feature, which you’ll learn about in Chapter 15, Templates, C++20 Concepts and Metaprogramming.

To implement the polymorphic behavior, the runtime concept idiom uses a clever mixture of class templates, inheritance and `private virtual` member functions to call the stand-alone overloaded `draw` functions. The templates, inheritance and `virtual` member functions are **implementation details** hidden from the client-code programmer.

Runtime Polymorphism with Type Erasure

C++ **type erasure**[53] is a template-based technique for implementing runtime polymorphism using **duck typing**. Templates enable you to remove specific types (that is, erase those types) from the code you write. Unlike using `std::variant` (Section 10.13), **type erasure does not require you to know in advance the types you’ll process polymorphically**. The runtime concept idiom uses type erasure under the hood.

53. Arthur O’Dwyer, “What Is Type Erasure?” March 18. 2019. Accessed January 11, 2022. https://quuxplusone.github.io/blog/2019/03/18/what-is-type-erasure/.

As type erasure has become more popular, various C++ open-source libraries have been developed to simplify writing runtime polymorphic programs that use **duck typing**. In particular,

- Facebook's open-source **Folly library**[54] includes a `Poly` **class template** for type-erasure-based runtime polymorphism[55] and

54. "Folly: Facebook Open-Source Library." Accessed January 11, 2022. https://github.com/face-book/folly.
55. "`folly/Poly.h`." Accessed January 11, 2022. https://github.com/facebook/folly/blob/master/folly/docs/Poly.md.

- Louis Dionne, a C++ templates guru, developed his own type-erasure-based runtime-polymorphism library called **Dyno**.[56,57]

56. "Dyno: Runtime Polymorphism Done Right." Accessed January 11, 2022. https://github.com/ldionne/dyno.
57. Lewis Dionne, "Runtime Polymorphism: Back to the Basics," YouTube video, November 5, 2017. Accessed January 12, 2022. https://www.youtube.com/watch?v=gVGtNFg4ay0.

Runtime Type Information (RTTI)

When processing `Employee` objects with runtime polymorphism in Section 10.9.4's `Employee` payroll example, we did not need to worry about the kind of `Employee` for which we were calculating the earnings. Sometimes with runtime polymorphism, you may need to temporarily treat an object as its actual type rather than its base-class type. For example, in our `Employee` payroll example, you might want to give `SalariedEmployee`s a bonus as you calculate the earnings for all `Employee`s. In Chapter 20, Other Topics and a Look Toward the Future of C++, we show a version of our `Employee` payroll example that demonstrates C++'s **runtime type information (RTTI)** and **dynamic casting**, which are used to determine an object's type at execution time.

10.17.2 Compile-Time (Static) Polymorphism Techniques

A significant Modern C++ theme is to do more at compile-time for better type-checking and better runtime performance.[58,59] We discuss various standard-library class templates in Chapter 13, Standard Library Containers and Iterators, and Chapter 14, Standard Library Algorithms and C++20 Ranges & Views. We discuss custom template programming in Chapter 15, Templates, C++20 Concepts and Metaprogramming.

58. "C++ Core Guidelines—Per: Performance." Accessed January 11, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-performance.
59. "Big Picture Issues—What's the Big Deal with Generic Programming?" Accessed January 11, 2022. https://isocpp.org/wiki/faq/big-picture#generic-paradigm.

Perf C++ programmers commonly use templates for compile-time generation of code that can handle many types—the code is generated on a type-by-type basis. In later chapters, we'll demonstrate custom class templates and continue discussing function templates—both are used to implement compile-time (static) polymorphism. With templates, compilers can identify errors before a program can execute and can perform optimizations for better runtime performance.[60] Here we briefly mention various compile-time polymorphism-related techniques.

60. "Compile Time vs Run Time Polymorphism in C++ Advantages/Disadvantages." Accessed January 11, 2022. https://stackoverflow.com/questions/16875989/compile-time-vs-run-time-polymorphism-in-c-advantages-disadvantages.

SFINAE: Substitution Failure Is Not an Error

Templates provide compile-time polymorphism by enabling you to generate from a class template or function template code that is specialized for use with a given type or types. Templates often have

requirements that types must satisfy for the compiler to generate valid code for those types. Consider our `maximum` function template in Section 5.16. It compared its arguments using the `>` operator, so our function template supports only types that work with that operator. If we provide type arguments to that function template that do not support the `>` operator, the compiler will generate an error message. **SFINAE (substitution failure is not an error)**[61,62,63] is a term describing how the compiler discards invalid template specialization code as it tries to determine the correct function to call. If the compiler finds a proper match for the call, the code is valid. This technique prevents the compiler from immediately generating (potentially) lengthy lists of error messages when it first tries to specialize a template. Chapter 15 revisits SFINAE in the context of C++20's concepts.[64,65]

61. "Substitution Failure Is Not an Error." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error.
62. Bartlomiej Filipek, "Notes on C++ SFINAE, Modern C++ and C++20 Concepts," April 20, 2020. Accessed January 11, 2022. https://www.bfilipek.com/2016/02/notes-on-c-sfinae.html.
63. David Vandevoorde and Nicolai M. Josuttis, *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

64. Marius Bancila, "Concepts versus SFINAE-Based Constraints," October 4, 2019. Accessed January 11, 2022. https://mariusbancila.ro/blog/2019/10/04/concepts-versus-sfinae-based-constraints/.
65. Bartlomiej Filipek, "Notes on C++ SFINAE, Modern C++ and C++20 Concepts."

20 C++20 Concepts

C++20's **concepts**[66,67,68] feature makes coding with templates simpler by enabling you to **constrain the types that can be used to instantiate templates**. The compiler can then check whether a type satisfies a template's constraints before using the template to generate code for that type. In Chapter 15, we'll show C++20's new concepts feature.[69]

66. Saar Raz, "C++20 Concepts: A Day in the Life," YouTube video, October 17, 2019. Accessed January 12, 2020. https://www.youtube.com/watch?v=qawSiMIXtE4.
67. "Constraints and Concepts." Accessed January 11, 2022. https://en.cppreference.com/w/cpp/language/constraints.
68. "Concepts Library." Accessed January 11, 2022. https://en.cppreference.com/w/cpp/concepts.
69. Bancila, "Concepts versus SFINAE-based constraints."

Tag Dispatch

20 **Tag dispatch**[70] is a template-based technique in which the compiler determines the version of an overloaded function to call based not only on template type parameters but also on properties of those types. Bjarne Stroustrup in his paper, "Concepts: The Future of Generic Programming," refers to properties of types as "concepts" and calls the tag-dispatching technique **concept overloading** or **concept-based overloading**.[71] He then discusses how C++20's new concepts feature can be used in place of tag dispatch.

70. "Generic Programming: Tag Dispatching." Accessed January 11, 2022. https://www.boost.org/community/generic_programming.html#tag_dispatching.
71. Bjarne Stroustrup, "Concepts: The Future of Generic Programming (Section 6)," January 31, 2017. Accessed January 11, 2022. https://www.stroustrup.com/good_concepts.pdf.

10.17.3 Other Polymorphism Concepts

You may also be interested in the following advanced resources:

- **Double dispatch**[72,73] (also called the **visitor pattern**[74]) is a runtime polymorphism technique that uses the runtime types of *two* objects to determine the correct member function to call. Double dispatch also can be implemented using `std::variant` and `std::visit`.[75]

72. "Double Dispatch." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Double_dispatch.
73. Barath Kannan, "Generalised Double Dispatch," YouTube video, October 26, 2019. Accessed January 12, 2022. https://www.youtube.com/watch?v=nNqiBasCab4.
74. "Visitor Pattern." Wikipedia. Wikimedia Foundation. Accessed January 11, 2022. https://en.wikipedia.org/wiki/Visitor_pattern.
75. Vishal Chovatiya. "Double Dispatch in C++: Recover Original Type of the Object Pointed by Base Class Pointer." April 11, 2020. Accessed January 11, 2022. http://www.vishalchovatiya.com/double-dispatch-in-cpp.

- In the presentation "Dynamic Polymorphism with Metaclasses and Code Injection" (CPPCON 2020), Sy Brand, Microsoft C++ Developer Advocate, presents advanced dynamic polymorphism techniques, using features that might be included in C++23 and C++26.[76]

76. Sy Brand, "Dynamic Polymorphism with Metaclasses and Code Injection," YouTube video, October 1, 2020. Accessed January 12, 2022. https://www.youtube.com/watch?v=8c6BAQcYF_E.

10.18 Wrap-Up

This chapter continued our object-oriented programming (OOP) discussion with introductions to inheritance and runtime polymorphism. You created derived classes that inherited base classes' capabilities, then customized or enhanced them. We distinguished between the *has-a* composition relationship and the *is-a* inheritance relationship.

We explained and demonstrated runtime polymorphism with inheritance hierarchies. You wrote programs that processed objects of classes that are part of the same class hierarchy as if they were all objects of the hierarchy's base class. You manipulated those objects via base-class pointers and references.

We explained that runtime polymorphism helps you design and implement systems that are easier to extend, enabling you to add new classes with little or no modification to the program's general portions. We used a detailed illustration to help you understand how runtime polymorphism, virtual functions and dynamic binding can be implemented "under the hood."

We introduced the non-virtual interface (NVI) idiom in which each base-class function serves only one purpose—as either a `public` non-`virtual` function that client code calls to perform a task or as a `private` (or `protected`) `virtual` function that derived classes can customize. You saw that when using this idiom, the `virtual` functions are internal implementation details hidden from the client code, making systems easier to maintain and evolve.

We initially focused on implementation inheritance, then pointed out that decades of experience with real-world, business-critical and mission-critical systems have shown that it can be difficult to maintain and modify such systems. So, we refactored our `Employee`-payroll example to use interface inheritance in which the base class contained only pure `virtual` functions that derived concrete

classes were required to implement. In that example, we introduced the composition-and-dependency-injection approach in which a class contains a pointer to an object that provides behaviors required by objects of the class. Then, we discussed how this interface-based approach makes the example easier to modify and evolve.

Next, we looked at another way to implement runtime polymorphism by processing objects of classes not related by inheritance, using duck typing via C++17's class template `std::variant` and the standard-library function `std::visit`.

We demonstrated multiple inheritance, discussed its potential problems and showed how `virtual` inheritance can solve them. We introduced the `protected` access specifier—derived-class member functions and `friend`s of the derived class can access `protected` base-class members. We also explained the three types of inheritance—`public`, `protected` and `private`—and the accessibility of base-class members in a derived class when using each type.

Finally, we overviewed other runtime-polymorphism techniques and several template-based compile-time polymorphism approaches. We also provided links to advanced resources for developers who wish to dig deeper.

A key goal of this chapter was to familiarize you with the mechanics of inheritance and runtime polymorphism with `virtual` functions. Now, you'll be able to better appreciate newer polymorphism idioms and techniques that can promote ease of modifiability and better performance. We cover some of these modern idioms in later chapters.

In Chapter 11, we continue our object-oriented programming presentation with operator overloading, which enables existing operators to work with custom class-type objects as well. For example, you'll see how to overload the `<<` operator to output a complete array without explicitly using an iteration statement. You'll use "smart pointers" to manage dynamically allocated memory and ensure that it's released when no longer needed. We'll introduce the remaining special member functions and discuss rules and guidelines for using them. We'll consider move semantics, which enable object resources (such as dynamically allocated memory) to move from one object to another when an object is going out of scope. As you'll see, this can save memory and increase performance.

11. Operator Overloading, Copy/Move Semantics and Smart Pointers

Objectives

In this chapter, you'll:

- Use built-in `string` class overloaded operators.
- Use operator overloading to help you craft valuable classes.
- Understand the special member functions and when to implement them for custom types.
- Understand when objects should be moved vs. copied.
- Use *rvalue* references and move semantics to eliminate unnecessary copies of objects that are going out of scope, improving program performance.
- Understand why you should avoid dynamic memory management with operators `new` and `delete`.
- Manage dynamic memory automatically with smart pointers.
- Craft a polished `MyArray` class that defines the five special member functions to support copy and move semantics, and overloads many unary and binary operators.
- Use C++20's three-way comparison operator (`<=>`).
- Convert objects to other types.
- Use the keyword `explicit` to prevent constructors and conversion operators from being used for implicit conversions.
- Experience a "light-bulb moment" when you'll truly appreciate the elegance and beauty of the class concept.

Outline

11.1 Introduction
11.2 Using the Overloaded Operators of Standard Library Class `string`
11.3 Operator Overloading Fundamentals
11.3.1 Operator Overloading Is Not Automatic
11.3.2 Operators That Cannot Be Overloaded
11.3.3 Operators That You Do Not Have to Overload
11.3.4 Rules and Restrictions on Operator Overloading
11.4 (Downplaying) Dynamic Memory Management with `new` and `delete`
11.5 Modern C++ Dynamic Memory Management: RAII and Smart Pointers
11.5.1 Smart Pointers
11.5.2 Demonstrating `unique_ptr`
11.5.3 `unique_ptr` Ownership
11.5.4 `unique_ptr` to a Built-In Array
11.6 `MyArray` Case Study: Crafting a Valuable Class with Operator Overloading
11.6.1 Special Member Functions
11.6.2 Using Class `MyArray`
11.6.3 `MyArray` Class Definition
11.6.4 Constructor That Specifies a `MyArray`'s Size
11.6.5 C++11 Passing a Braced Initializer to a Constructor
11.6.6 Copy Constructor and Copy Assignment Operator
11.6.7 Move Constructor and Move Assignment Operator
11.6.8 Destructor

11.6.9 `toString` and `size` Functions
11.6.10 Overloading the Equality (`==`) and Inequality (`!=`) Operators
11.6.11 Overloading the Subscript (`[]`) Operator
11.6.12 Overloading the Unary `bool` Conversion Operator
11.6.13 Overloading the Preincrement Operator
11.6.14 Overloading the Postincrement Operator
11.6.15 Overloading the Addition Assignment Operator (`+=`)
11.6.16 Overloading the Binary Stream Extraction (`>>`) and Stream Insertion (`<<`) Operators
11.6.17 `friend` Function `swap`
11.7 C++20 Three-Way Comparison Operator (`<=>`)
11.8 Converting Between Types
11.9 `explicit` Constructors and Conversion Operators
11.10 Overloading the Function Call Operator `()`
11.11 Wrap-Up

11.1 Introduction

This chapter presents **operator overloading**, which enables C++'s existing operators to work with custom class objects. One example of an overloaded operator in standard C++ is `<<`, which is used both as

- the stream insertion operator and
- the bitwise left-shift operator (discussed in Appendix E).

Similarly, `>>` is used both as

- the stream extraction operator and
- the bitwise right-shift operator (discussed in Appendix E).

You've already used many overloaded operators. Some are built into the core C++ language itself. For example, the `+` operator performs differently, based on its context in integer, floating-point and pointer arithmetic with data of fundamental types. Class `string` also overloads `+` so you can concatenate `string`s.

You can overload most operators for use with custom class objects. The compiler generates the appropriate code based on the operand types. The jobs performed by overloaded operators also can be performed by explicit function calls, but operator notation is often more natural and more convenient.

`string` Class Overloaded Operators Demonstration

In Section 2.7's Objects-Natural case study, we introduced the standard library's `string` class and demonstrated a few of its features, such as string concatenation with the `+` operator. Here, we demonstrate many additional `string`-class overloaded operators, so you can appreciate how valuable operator overloading is in a key standard library class before implementing it in your own custom classes. Next, we'll present operator-overloading fundamentals.

Dynamic Memory Management and Smart Pointers

We introduce dynamic memory management, which enables a program to acquire the additional memory it needs for objects at runtime rather than at compile-time and release that memory when it's no longer needed so it can be used for other purposes. We discuss the potential problems with dynamically allocated memory, such as forgetting to release memory that's no longer needed—known as a **memory leak**. Then, we introduce smart pointers, which can automatically release

dynamically allocated memory for you. As you'll see, when coupled with the **RAII (Resource Acquisition Is Initialization) strategy**, **smart pointers** enable you to eliminate subtle memory leak issues.

`MyArray` Case Study

Next, we present one of the book's capstone case studies. We build a custom `MyArray` class that uses overloaded operators and other capabilities to solve various problems with C++'s native pointer-based arrays. We introduce and implement the five **special member functions** you can define in each class—the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor. Sometimes the default constructor also is included as a special member function. We introduce copy semantics and move semantics, which help tell a compiler when it can move resources from one object to another to avoid costly unnecessary copies.

`MyArray` uses smart pointers and RAII, and overloads many unary and binary operators, including

- `=` (assignment),
- `==` (equality),
- `!=` (inequality),
- `[]` (subscript),
- `++` (increment),
- `+=` (addition assignment),
- `>>` (stream extraction) and
- `<<` (stream insertion).

We also show how to define a conversion operator that converts a `MyArray` to a `true` or `false bool` value, indicating whether a `MyArray` contains elements or is empty.

Many of our readers have said that working through the `MyArray` case study is a "light bulb moment," helping them truly appreciate what classes and object technology are all about. Once you master this `MyArray` class, you'll indeed understand the essence of object technology—**crafting, using and reusing valuable classes, and sharing them with colleagues and perhaps the entire C++ open-source community**.

C++20's Three-Way Comparison Operator (`<=>`)

20 We introduce C++20's new three-way comparison operator (<=>), which is also referred to as the "spaceship operator."[1]

---

1. "Spaceship operator" was coined by Randal L. Schwartz when he was teaching the same operator in a Perl programming course—the operator reminded him of a spaceship in an early video game. https://groups.google.com/a/dartlang.org/g/misc/c/WS5xftItpl4/m/jcIttrMq8agJ?pli=1.

Conversion Operators

We discuss overloaded conversion operators and conversion constructors in more depth. In particular, we demonstrate how implicit conversions can cause subtle problems. Then, we use `explicit` to prevent those problems.

11.2 Using the Overloaded Operators of Standard Library Class `string`

Chapter 8 presented class `string`. Figure 11.1 demonstrates many of `string`'s overloaded operators and several other useful member functions, including `empty`, `substr` and `at`:

- `empty` determines whether a `string` is empty,

- `substr` (for “substring”) returns a portion of an existing `string` and
- `at` returns the character at a specific index in a `string` (after checking that the index is in range).

For discussion purposes, we split this example into small chunks of code, each followed by its output.

```
1  // fig11_01.cpp
2  // Standard library string class test program.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <string>
6  #include <string_view>
7
8  int main() {
```

Fig. 11.1 Standard library `string` class test program.

Creating `string` and `string_view` Objects and Displaying Them with `cout` and Operator `<<`

Lines 9–12 create three `string`s and a `string_view`:

- the `string s1` is initialized with the literal `"happy"`,
- the `string s2` is initialized with the literal `" birthday"`,
- the `string s3` uses `string`’s default constructor to create an empty `string` and
- the `string_view v` is initialized to reference the characters in the literal `"hello"`.

Lines 15–16 output these three objects, using `cout` and operator `<<`, which `string` and `string_view` overload for output purposes.

```
9  std::string s1{"happy"}; // initialize string from char*
10 std::string s2{" birthday"}; // initialize string from char*
11 std::string s3; // creates an empty string
12 std::string_view v{"hello"}; // initialize string_view from char*
13
14 // output strings and string_view
15 std::cout << "s1: \"" << s1 << "\"; s2: \"" << s2
16    << "\"; s3: \"" << s3 << "\"; v: \"" << v << "\"\n\n";
17
```

```
s1: "happy"; s2: " birthday"; s3: ""; v: "hello"
```

Comparing `string` Objects with the Equality and Relational Operators

20 Lines 19–25 show the results of comparing `s2` to `s1` using class `string`’s overloaded equality and relational operators. These perform lexicographical comparisons using the numerical values of the characters in each `string` (see Appendix B, Character Set). When you use C++20’s `format` function to convert a `bool` to a `string`, `format` produces `"true"` or `"false"`.

```
18 // test overloaded equality and relational operators
19 std::cout << "The results of comparing s2 and s1:\n"
20    << fmt::format("s2 == s1: {}\n", s2 == s1)
21    << fmt::format("s2 != s1: {}\n", s2 != s1)
22    << fmt::format("s2 > s1: {}\n", s2 > s1)
23    << fmt::format("s2 < s1: {}\n", s2 < s1)
```

```
24       << fmt::format("s2 >= s1: {}\n", s2 >= s1)
25       << fmt::format("s2 <= s1: {}\n\n", s2 <= s1);
26
```

```
The results of comparing s2 and s1:
s2 == s1: false
s2 != s1: true
s2 > s1: false
s2 < s1: true
s2 >= s1: false
s2 <= s1: true
```

`string` Member Function `empty`

Line 30 uses `string` member function **`empty`**, which returns `true` if the `string` is empty; otherwise, it returns `false`. The object `s3` was initialized with the default constructor, so it is empty.

```
27    // test string member function empty
28    std::cout << "Testing s3.empty():\n";
29
30    if (s3.empty()) {
31       std::cout << "s3 is empty; assigning s1 to s3;\n";
32       s3 = s1; // assign s1 to s3
33       std::cout << fmt::format("s3 is \"{}\"\n\n", s3);
34    }
35
```

```
Testing s3.empty():
s3 is empty; assigning s1 to s3;
s3 is "happy"
```

`string` Copy Assignment Operator

Line 32 demonstrates class `string`'s **overloaded copy assignment operator** by assigning `s1` to `s3`. Line 33 outputs `s3` to demonstrate that the assignment worked correctly.

`string` Concatenation and C++14 `string`-Object Literals

14 Line 37 demonstrates `string`'s overloaded `+=` operator for **string concatenation assignment**. In this case, the contents of `s2` are appended to `s1`, thus modifying its value. Then line 38 outputs the updated `s1`. Line 41 demonstrates that you also may append a C-string literal to a `string` object using operator `+=`. Line 42 displays the result. Similarly, line 46 concatenates `s1` with a C++14 **`string`-object literal**, which is indicated by placing the letter `s` immediately after the closing `"` of a `string` literal, as in

```
", have a great day!"s
```

The preceding literal actually calls a C++ standard library function that returns a `string` object containing the literal's characters. Lines 47–48 display `s1`'s new value.

```
36    // test overloaded string concatenation assignment operator
37    s1 += s2; // test overloaded concatenation
38    std::cout << fmt::format("s1 += s2 yields s1 = {}\n\n", s1);
39
```

```
40   // test string concatenation with a C string
41   s1 += " to you";
42   std::cout << fmt::format("s1 += \" to you\" yields s1 = {}\n\n", s1);
43
44   // test string concatenation with a C++14 string-object literal
45   using namespace std::string_literals;
46   s1 += ", have a great day!"s; // s after " for string-object literal
47   std::cout << fmt::format(
48      "s1 += \", have a great day!\"s yields\ns1 = {}\n\n", s1);
49
```

```
s1 += s2 yields s1 = happy birthday

s1 += " to you" yields s1 = happy birthday to you

s1 += ", have a great day!"s yields
s1 = happy birthday to you, have a great day!
```

`string` Member Function `substr`

Class `string` provides member function **`substr`** (lines 53 and 58) to return a `string` containing a portion of the `string` object on which the function is called. The call in line 53 obtains a 14-character substring of `s1` starting at position 0. Line 58 obtains a substring starting from position 15 of `s1`. When the second argument is not specified, `substr` returns the remainder of the `string` on which it's called.

```
50   // test string member function substr
51   std::cout << fmt::format("{} {}\n{}\n\n",
52      "The substring of s1 starting at location 0 for",
53      "14 characters, s1.substr(0, 14), is:", s1.substr(0, 14));
54
55   // test substr "to-end-of-string" option
56   std::cout << fmt::format("{} {}\n{}\n\n",
57      "The substring of s1 starting at",
58      "location 15, s1.substr(15), is:", s1.substr(15));
59
```

```
The substring of s1 starting at location 0 for 14 characters,
s1.substr(0, 14), is:
happy birthday

The substring of s1 starting at location 15, s1.substr(15), is:
to you, have a great day!
```

`string` Copy Constructor

Line 61 creates `string` object `s4`, initializing it with a copy of `s1`. This calls class `string`'s **copy constructor**, which copies the contents of `s1` into the new object `s4`. You'll see how to define a copy constructor for a custom class in Section 11.6.

```
60   // test copy constructor
61   std::string s4{s1};
62   std::cout << fmt::format("s4 = {}\n\n", s4);
63
```

```
s4 = happy birthday to you, have a great day!
```

Testing Self-Assignment with the `string` Copy Assignment Operator

Line 66 uses class `string`'s overloaded copy assignment (`=`) operator to demonstrate that it handles **self-assignment** properly, so `s4` still has the same value after the self-assignment. We'll see when we build class `MyArray` later in the chapter that **self-assignment must be handled carefully for objects that manage their own memory**, and we'll show how to deal with the issues.

```
64   // test overloaded copy assignment (=) operator with self-assignment
65   std::cout << "assigning s4 to s4\n";
66   s4 = s4;
67   std::cout << fmt::format("s4 = {}\n\n", s4);
68
```

```
assigning s4 to s4
s4 = happy birthday to you, have a great day!
```

Initializing a `string` with a `string_view`

Line 71 demonstrates `string`'s constructor that receives a `string_view`, in this case, copying the character data represented by the `string_view v` (line 12) into the new `string s5`.

```
69   // test string's string_view constructor
70   std::cout << "initializing s5 with string_view v\n";
71   std::string s5{v};
72   std::cout << fmt::format("s5 is {}\n\n", s5);
73
```

```
initializing s5 with string_view v
s5 is hello
```

`string`'s `[]` Operator

Lines 75–76 use `string`'s overloaded `[]` operator in assignments to create *lvalues* for replacing characters in `s1`. Lines 77–78 output `s1`'s new value. The `[]` operator returns the character at the specified location as a modifiable *lvalue* or a nonmodifiable *lvalue* (e.g., a `const` reference), depending on the context in which the expression appears. For example:

- If `[]` is used on a non-`const string`, the function returns a modifiable *lvalue*, which can be used on the left of an assignment (`=`) operator to assign a new value to that location in the `string`, as in lines 77–78.
- If `[]` is used on a `const string`, the function returns a nonmodifiable *lvalue* that can be used to obtain, but not modify, the value at that location in the `string`.

The overloaded `[]` operator does not perform bounds checking. So, you must ensure that operations using this operator do not accidentally manipulate elements outside the `string`'s bounds.

```
74   // test using overloaded subscript operator to create lvalue
75   s1[0] = 'H';
76   s1[6] = 'B';
77   std::cout << fmt::format("{}:\n{}\n\n",
78      "after s1[0] = 'H' and s1[6] = 'B', s1 is",   s1);
79
```

```
after s1[0] = 'H' and s1[6] = 'B', s1 is:
```

```
Happy Birthday to you, have a great day!
```

`string`'s `at` Member Function

Class `string`'s **`at` member function** throws an exception if its argument is an invalid out-of-bounds index. If the index is valid, function `at` returns the character at the specified location as a modifiable *lvalue* or a nonmodifiable *lvalue* (e.g., a `const` reference), depending on the context in which the call appears. Line 83 demonstrates a call to function `at` with an invalid index causing an `out_of_range` exception. Here, we show the error message produced by GNU C++.

```
80       // test index out of range with string member function "at"
81       try{
82          std::cout << "Attempt to assign 'd' to s1.at(100) yields:\n";
83          s1.at(100) = 'd'; // ERROR: subscript out of range
84       }
85       catch (const std::out_of_range& ex) {
86          std::cout << fmt::format("An exception occurred: {}\n", ex.what());
87       }
88    }
```

```
Attempt to assign 'd' to s1.at(100) yields:
An exception occurred: basic_string::at: __n (which is 100) >= this->size()
(which is 40)
```

11.3 Operator Overloading Fundamentals

SE As you saw in Fig. 11.1, standard library class `string`'s overloaded operators provide a concise notation for manipulating `string` objects. You can use operators with your own user-defined types as well. C++ allows most existing operators to be overloaded by defining **`operator functions`**. Once you define an `operator` function for a given operator and your custom class, that operator has meaning appropriate for objects of your class.

11.3.1 Operator Overloading Is Not Automatic

SE You must write `operator` functions that perform the desired operations. An operator is overloaded by writing a non-`static` member function definition or a non-member function definition. An operator function's name is the keyword **`operator`**, followed by the symbol of the operator being overloaded. For example, the function name `operator+` would overload the addition operator (`+`). **When operators are overloaded as member functions, they must be non-`static`. They are called on an object of the class and operate on that object.**

11.3.2 Operators That Cannot Be Overloaded

Most of C++'s operators can be overloaded—the following operators cannot:[2]

2. Although it's possible to overload the address (`&`), comma (`,`), `&&` and `||` operators, you should avoid doing so to avoid subtle errors. See https://isocpp.org/wiki/faq/operator-overloading. Accessed January 15, 2022.

- `.` (dot) member-selection operator

- `.*` pointer-to-member operator (discussed in Section 20.6)
- `::` scope-resolution operator
- `?:` conditional operator—though this might be overloadable in the future[3]

3. Matthias Kretz, “Making Operator `?:` Overloadable,” October 7, 2019. Accessed January 15, 2022. https://wg21.link/p0917.

11.3.3 Operators That You Do Not Have to Overload

Three operators work with objects of each new class by default:

- Err 11 The **assignment operator (`=`)** may be used with *most* classes to perform **member-wise assignments** of the data members. The default assignment operator assigns each data member from the “source” object (on the right) to the “target” object (on the left). As you’ll see in Section 11.6.6, **this can be dangerous for classes that have pointer members**. So, you’ll either explicitly overload the assignment operator or explicitly disallow the compiler from defining the default assignment operator. This is also true for the **C++11 move assignment operator**, which we discuss in Section 11.6.
- The **address (`&`) operator** returns a pointer to the object.
- The **comma operator** evaluates the expression to its left, then the expression to its right, and returns the latter expression’s value. Though this operator can be overloaded, generally it is not.[4]

4. “Operator Overloading.” Accessed January 15, 2022. https://isocpp.org/wiki/faq/operator-overloading.

11.3.4 Rules and Restrictions on Operator Overloading

As you prepare to overload operators for your own classes, there are several rules and restrictions you should keep in mind:

- **An operator’s precedence cannot be changed by overloading.** Parentheses can be used to change the order of evaluation of overloaded operators in an expression.
- **An operator’s grouping cannot be changed by overloading**. If an operator normally groups left-to-right, then so do its overloaded versions.
- **An operator’s “arity” (the number of operands an operator takes) cannot be changed by overloading**. Overloaded unary operators remain unary operators, and overloaded binary operators remain binary operators. The only ternary operator (`?:`) cannot be overloaded.
Operators `&`, `*`, `+` and `-` each have unary and binary versions that can be overloaded separately.
- **Only existing operators can be overloaded**. You cannot create new ones.
- **You cannot overload operators to change how an operator works on only fundamental-type values**. For example, you cannot make `+` subtract two `int`s.
- Operator overloading works only with **objects of user-defined types** or with a **mixture of an object of a user-defined type and an object of a fundamental type**.
- 20 **Related operators, like `+` and `+=`, generally must be overloaded separately**. In C++20, if you define `==` for your class, C++ provides `!=` for you—it simply returns the opposite of `==`.
- **When overloading `()`, `[]`, `->` or `=`, the operator overloading function must be declared as a class member**. You’ll see later that this is required when the left operand must be an object of your custom class type. Operator functions for all other overloadable operators may be member or non-member functions.

SE SE You should overload operators for class types to work as closely as possible to how they work with fundamental types. Avoid excessive or inconsistent use of operator overloading, as this can make a program cryptic and difficult to read.

11.4 (Downplaying) Dynamic Memory Management with `new` and `delete`

CG You can **allocate** and **deallocate** memory in a program for objects and for arrays of any built-in or user-defined type. This is known as **dynamic memory management**. In Chapter 7, we introduced pointers and showed various old-style techniques you're likely to see in legacy code, then we showed improved Modern C++ techniques. We do the same here. For decades, C++ dynamic memory management was performed with operators `new` and `delete`. **The C++ Core Guidelines recommend against using these operators directly.**[5,6] You'll likely see them in legacy C++ code, so we discuss them here. Section 11.5 discusses Modern C++ dynamic memory management techniques, which we'll use in Section 11.6's `MyArray` case study.

5. C++ Core Guidelines, "R.11: Avoid Calling `new` and `delete` Explicitly." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-newdelete.
6. Operators `new` and `delete` can be overloaded. If you overload `new`, you should overload `delete` in the same scope to avoid subtle dynamic memory management errors. Overloading `new` and `delete` is typically done for precise control over how memory is allocated and deallocated, often for performance. This might be used, for example, to preallocate a pool of memory, then create new objects within that pool to reduce runtime memory-allocation overhead. For an overview of the placement `new` and `delete` operators, see https://en.wikipedia.org/wiki/Placement_syntax.

The Old Way: Operators `new` and `delete`

You can use the `new` operator to dynamically reserve (that is, allocate) the exact amount of memory required to hold an object or built-in array. The object or built-in array is created on the **free store**—a region of memory assigned to each program for storing dynamically allocated objects. Once memory is allocated, you can access it via the pointer returned by operator `new`. When you no longer need the memory, you can return it to the free store by using the `delete` operator to deallocate (i.e., release) the memory, which can then be reused by future `new` operations.

Obtaining Dynamic Memory with `new`

Consider the following statement:

```
Time* timePtr{new Time{}};
```

Err The `new` operator allocates memory for a `Time` object, calls a default constructor to initialize it and returns a `Time*`—a pointer of the type specified to the right of the `new` operator. The preceding statement calls `Time`'s default constructor because we did not supply constructor arguments. If `new` is unable to find sufficient space in memory for the object, it throws a **`bad_alloc` exception**. Chapter 12 shows how to deal with `new` failures.

Releasing Dynamic Memory with `delete`

To destroy a dynamically allocated object and free the space for the object, use `delete`:

```
delete timePtr;
```

Err This calls the destructor for the object to which `timePtr` points, then deallocates the object's memory, returning it to the free store. Not releasing dynamically allocated memory when it's no longer needed can cause memory leaks that eventually lead to a system running out of memory prematurely. Sometimes the problem might be even worse. If the leaked memory contains objects that manage other resources, those objects' destructors will not be called to release the resources, causing additional leaks.

Err Err Do not `delete` memory that was not allocated by `new`. Doing so results in **undefined behavior**. After you `delete` dynamically allocated memory, be sure not to `delete` the same memory again, as this typically causes a program to crash. One way to guard against this is to immediately set the pointer to `nullptr`—deleting such a pointer has no effect.

Initializing Dynamically Allocated Objects

You can initialize a newly allocated object with constructor arguments, as in

```
Time* timePtr{new Time{12, 45, 0}};
```

which initializes a new `Time` object to 12:45:00 PM and assigns its pointer to `timePtr`.

Dynamically Allocating Built-In Arrays with `new[]`

You also can use the `new` operator to dynamically allocate built-in arrays. The following statement dynamically allocates a 10-element built-in array of `int`s:

```
int* gradesArray{new int[10]{}};
```

This statement aims the `int` pointer `gradesArray` at the first element of the dynamically allocated array. The empty braced initializer following `new int[10]` value initializes the array's elements, which sets fundamental-type elements to `0`, `bool`s to `false` and pointers to `nullptr`. The braced initializer may also contain a comma-separated list of initializers for the array's elements. Value initializing an object calls its default constructor, if available. The rules become more complicated for objects that do not have default constructors. For more details, see the value-initialization rules at:

```
https://en.cppreference.com/w/cpp/language/value_initialization
```

The size of a built-in array created at compile-time must be specified using an integral constant expression. However, a dynamically allocated array's size can be specified using *any* non-negative integral expression.

Releasing Dynamically Allocated Built-In Arrays with `delete[]`

To deallocate the memory to which `gradesArray` points, use the statement

```
delete[] gradesArray;
```

If the pointer points to a built-in array of objects, this statement first calls the destructor for each object in the array, then deallocates the memory for the entire array. As with `delete`, `delete[]` on a `nullptr` has no effect.

Err **Using `delete` instead of `delete[]` for an array allocated with `new[]` results in undefined behavior.** Some compilers call the destructor only for the first object in the array. To ensure that every object in the array receives a destructor call, **`always use operator delete[]` to delete memory allocated by `new[]`.** We'll show better techniques for managing dynamically allocated memory that enable you to avoid using `new` and `delete`.

Err If there is only one pointer to a block of dynamically allocated memory and the pointer goes out of scope, or if you assign it `nullptr` or a different memory address, **a memory leak occurs**.

Err **After deleting dynamically allocated memory, set the pointer's value to `nullptr`** to indicate that it no longer points to memory in the free store. This ensures that your code cannot inadvertently access the previously allocated memory—doing so could cause subtle logic errors.

Range-Based `for` Does Not Work with Dynamically Allocated Built-In Arrays

Err You might be tempted to use C++11's range-based `for` statement to iterate over dynamically allocated arrays. Unfortunately, this will not compile. The compiler must know the number of elements at compile-time to iterate over an array with range-based `for`. As a workaround, you can create a C++20 `span` object that represents the dynamically allocated array and its number of elements, then iterate over the `span` object with range-based `for`.

11.5 Modern C++ Dynamic Memory Management: RAII and Smart Pointers

A common design pattern is to

- allocate dynamic memory,
- assign the address of that memory to a pointer,
- use the pointer to manipulate the memory and
- deallocate the memory when it's no longer needed.

Err If an exception occurs after successful memory allocation but before the `delete` or `delete[]` statement executes, a **memory leak** could occur.

CG For this reason, the C++ Core Guidelines recommend that you manage resources like dynamic memory using **RAII—Resource Acquisition Is Initialization**.[7,8] The concept is straightforward. For any resource that must be returned to the system when the program is done using it, the program should

7. C++ Core Guidelines, "R: Resource Management." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-resource.
8. C++ Core Guidelines, "R.1: Manage Resources Automatically Using Resource Handles and RAII (Resource Acquisition Is Initialization)." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-raii.

- create the object as a local variable in a function—**the object's constructor should acquire the resource while initializing the object**,

- use that object as necessary in your program, then
- when the function call terminates, the object goes out of scope—**the object's destructor should release the resource**.

11.5.1 Smart Pointers

SE C++11 **smart pointers** use RAII to manage dynamically allocated memory for you. The standard library header `<memory>` defines three smart pointer types:

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

A `unique_ptr` maintains a pointer to **dynamically allocated memory that can belong to only one `unique_ptr` at a time**. When a `unique_ptr` object goes out of scope, its destructor uses `delete` (or `delete[]` for an array) to deallocate the memory that the `unique_ptr` manages. The rest of Section 11.5 demonstrates `unique_ptr`, which we'll also use in our `MyArray` case study. We introduce `shared_ptr` and `weak_ptr` in online Chapter 20, Other Topics and a Look Toward the Future of C++.

11.5.2 Demonstrating `unique_ptr`

14 Figure 11.2 demonstrates a `unique_ptr` pointing to a dynamically allocated object of class `Integer` (lines 7–22). For pedagogic purposes, the class's constructor and destructor both display when they are called. Line 29 creates `unique_ptr` object `ptr` and initializes it with a pointer to a dynamically allocated `Integer` object that contains the value `7`. To initialize the `unique_ptr`, line 29 calls C++14's **`make_unique` function template**, which allocates dynamic memory with operator `new`, then returns a `unique_ptr` to that memory.[9] In this example, `make_unique<Integer>` returns a `unique_ptr<Integer>`—line 29 uses the `auto` keyword to infer `ptr`'s type from its initializer.

9. Prior to C++14, you'd pass the result of a `new` expression directly to `unique_ptr`'s constructor.

```
1  // fig11_02.cpp
2  // Demonstrating unique_ptr.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <memory>
6
7  class Integer {
8  public:
9     // constructor
10    Integer(int i) : value{i} {
11       std::cout << fmt::format("Constructor for Integer {}\n", value);
12    }
13
14    // destructor
15    ~Integer() {
16       std::cout << fmt::format("Destructor for Integer {}\n", value);
17    }
18
19    int getValue() const {return value;} // return Integer value
20 private:
21    int value{0};
```

```
   };

   // use unique_ptr to manipulate Integer object
   int main() {
      std::cout << "Creating a unique_ptr that points to an Integer\n";

      // create a unique_ptr object and "aim" it at a new Integer object
      auto ptr{std::make_unique<Integer>(7)};

      // use unique_ptr to call an Integer member function
      std::cout << fmt::format("Integer value: {}\n\nMain ends\n",
         ptr->getValue());
   }
```

```
Creating a unique_ptr that points to an Integer
Constructor for Integer 7
Integer value: 7

Main ends
Destructor for Integer 7
```

Fig. 11.2 Demonstrating `unique_ptr`.

Line 33 uses `unique_ptr`'s overloaded `->` operator to invoke function `getValue` on the `Integer` object that `ptr` manages. The expression `ptr->getValue()` also could have been written as

```
(*ptr).getValue()
```

which uses `unique_ptr`'s overloaded `*` operator to dereference `ptr`, then uses the dot (`.`) operator to invoke function `getValue` on the `Integer` object.

SE Because `ptr` is a local variable in `main`, it's destroyed when `main` terminates. The `unique_ptr` destructor `delete`s the dynamically allocated `Integer` object, which calls the object's destructor. **The program releases the `Integer` object's memory, whether program control leaves the block normally via a `return` statement or reaching the end of the block—or as the result of an exception.**

SE Most importantly, using `unique_ptr` **prevents resource leaks**. For example, suppose a function returns a pointer aimed at some dynamically allocated object. Unfortunately, the function caller that receives this pointer might not `delete` the object, resulting in a **memory leak**. However, if the function returns a `unique_ptr` to the object, the object will be deleted automatically when the `unique_ptr` object's destructor gets called.

11.5.3 `unique_ptr` Ownership

SE Only one `unique_ptr` can own a dynamically allocated object, so assigning one `unique_ptr` to another **transfers ownership** to the target `unique_ptr`. The same is true when one `unique_ptr` is passed as an argument to another `unique_ptr`'s constructor. These operations use `unique_ptr`'s move assignment operator and move constructor, which we discuss in Section 11.6. **The last `unique_ptr` object that owns the dynamic memory will delete the memory. This makes `unique_ptr` an ideal mechanism for returning ownership of dynamically allocated objects to client code.**

11.5.4 `unique_ptr` to a Built-In Array

You can also use a `unique_ptr` to manage a dynamically allocated built-in array, as we'll do in Section 11.6's `MyArray` case study. For example, in the statement

```
auto ptr{make_unique<int[]>(10)};
```

14 `make_unique`'s type argument is specified as `int[]`. So `make_unique` dynamically allocates a built-in array with the number of elements specified by its argument (`10`). By default, the `int` elements are value initialized to 0. The preceding statement uses `auto` to infer `ptr`'s type (`unique_ptr<int[]>`), based on its initializer.

A `unique_ptr` that manages an array provides an **overloaded subscript operator (`[]`)** to access its elements. For example, the statement

```
ptr[2] = 7;
```

assigns `7` to the `int` at `ptr[2]`, and the following statement displays that `int`:

```
std::cout << ptr[2] << "\n";
```

11.6 `MyArray` Case Study: Crafting a Valuable Class with Operator Overloading

Class development is an interesting, creative and intellectually challenging activity—always with the goal of crafting valuable classes. When we refer to "arrays" in this case study, we mean the built-in arrays discussed in Chapter 7. These pointer-based arrays have many problems, including:

- C++ does not check whether an array index is out-of-bounds. A program can easily "walk off" either end of an array, likely causing a fatal runtime error if you forget to test for this possibility in your code.
- Arrays of size n must use index values in the range 0 to $n - 1$. Alternative index ranges are not allowed.
- You cannot input an entire array with the stream extraction operator (`>>`) or output one with the stream insertion operator (`<<`). You must read or write every element.[10]

10. In Chapter 13, Standard Library Containers and Iterators, you'll use C++ standard library functions to input and output entire containers of elements, such as `vectors` and `arrays`.

- Two arrays cannot be meaningfully compared with equality or relational operators. Array names are simply pointers to where the arrays begin in memory. Two arrays will always be at different memory locations.
- 20 When you pass an array to a general-purpose function that handles arrays of any size, you must pass the array's size as an additional argument. As you saw in Section 7.10, C++20's `span`s help solve this problem.
- You cannot assign one array to another with the assignment operator(s).

With C++, you can implement more robust array capabilities via classes and operator overloading, as has been done with C++ standard library class templates `array` and `vector`. In this section, we'll develop our own custom `MyArray` class that's preferable to arrays. Internally, class `MyArray` will use a `unique_ptr` smart pointer to manage a dynamically allocated built-in array of `int` values.[11]

11. In this section, we'll use operator overloading to craft a valuable class. In Chapter 15, we'll make it more valuable by converting it to a class template, and we'll use C++20's new Concepts feature to add even more value.

We'll create a powerful `MyArray` class with the following capabilities:

- `MyArray`s perform range checking when you access them via the subscript (`[]`) operator to ensure indices remain within their bounds. Otherwise, the `MyArray` object will throw a standard library `out_of_bounds` exception.
- Entire `MyArray`s can be input or output with the overloaded stream extraction (`>>`) and stream insertion (`<<`) operators, without the client-code programmer having to write iteration statements.
- `MyArray`s may be compared to one another with the equality operators `==` and `!=`. The class could easily be enhanced to include the relational operators.
- `MyArray`s know their own size, making it easier to pass them to functions.
- `MyArray` objects may be assigned to one another with the assignment operator.
- `MyArray`s may be converted to `bool` (`false` or `true`) values to determine whether they are empty or contain elements.
- `MyArray` provides prefix and postfix increment (`++`) operators that add `1` to every element. We can easily add prefix and postfix decrement (`--`) operators.
- `MyArray` provides an addition assignment operator (`+=`) that adds a specified value to every element. The class could easily be enhanced to support the `-=`, `*=`, `/=` and `%=` assignment operators.

Class `MyArray` will demonstrate the **five special member functions** and the **`unique_ptr` smart pointer for managing dynamically allocated memory**. We'll use **RAII (Resource Acquisition Is Initialization)** throughout this example to manage the dynamically allocated memory resources. The class's constructors will dynamically allocate memory as `MyArray` objects are initialized. The class's destructor will deallocate the memory when the objects go out of scope**, preventing memory leaks.** Our `MyArray` class is not meant to replace standard library class templates `array` and `vector`, nor is it meant to mimic their capabilities. It demonstrates key C++ language and library features that you'll find useful when you build your own classes.

11.6.1 Special Member Functions

Every class you define can have five **special member functions**, each of which we define in class `MyArray`:

- a **copy constructor**,
- a **copy assignment operator**,
- a **move constructor**,
- a **move assignment operator** and
- a **destructor**.

The copy constructor and copy assignment operator implement the class's **copy semantics**—that is, how to copy a `MyArray` when it is passed by value to a function, returned by value from a function or assigned to another `MyArray`. The move constructor and move assignment operator implement the class's **move semantics**, which eliminate costly unnecessary copies of objects that are about to be destroyed. We discuss the details of these special member functions as we encounter the need for them throughout this case study.

11.6.2 Using Class `MyArray`

The program of Figs. 11.3–11.5 demonstrates class `MyArray` and its rich selection of overloaded operators. The code in Fig. 11.3 tests the various `MyArray` capabilities. We present the class definition in Fig. 11.4 and its member-function definitions in Fig. 11.5. We've broken the code and outputs into small segments for discussion purposes. For pedagogic purposes, many of class `MyArray`'s member functions, including all its special member functions, display output to show when they're called.

```
1  // fig11_03.cpp
2  // MyArray class test program.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <stdexcept>
6  #include <utility> // for std::move
7  #include "MyArray.h"
8
9  // function to return a MyArray by value
10 MyArray getArrayByValue() {
11    MyArray localInts{10, 20, 30}; // create three-element MyArray
12    return localInts; // return by value creates an rvalue
13 }
14
```

Fig. 11.3 `MyArray` class test program.

Function `getArrayByValue`

Later in this program, we'll call the `getArrayByValue` function (Fig. 11.3, lines 10–13) to create a local `MyArray` object by calling `MyArray`'s constructor that receives an **initializer list**. Function `getArrayByValue` returns that local object *by value*.

Creating `MyArray` Objects and Displaying Their Size and Contents

Lines 16–17 create objects `ints1` with seven elements and `ints2` with 10 elements. Each calls the `MyArray` constructor that receives the number of elements and initializes the elements to zeros. Lines 20–21 display `ints1`'s `size`, then output its contents using `MyArray`'s **overloaded stream insertion operator (`<<`)**. Lines 24–25 do the same for `ints2`.

```
15 int main() {
16    MyArray ints1(7); // 7-element MyArray; note () rather than {}
17    MyArray ints2(10); // 10-element MyArray; note () rather than {}
18
19    // print ints1 size and contents
20    std::cout << fmt::format("\nints1 size: {}\ncontents: ", ints1.size())
21       << ints1; // uses overloaded <<
22
23    // print ints2 size and contents
24    std::cout << fmt::format("\nints2 size: {}\ncontents: ", ints2.size())
25       << ints2; // uses overloaded <<
26
```

```
MyArray(size_t) constructor
MyArray(size_t) constructor
```

```
ints1 size: 7
contents: {0, 0, 0, 0, 0, 0, 0}

ints2 size: 10
contents: {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
```

Using Parentheses Rather Than Braces to Call the Constructor

So far, we've used braced initializers, `{}`, to pass arguments to constructors. Lines 16–17 use parentheses, `()`, to call the `MyArray` constructor that receives a size. We do this because our class—like the standard library's `array` and `vector` classes—also supports constructing a `MyArray` from a braced-initializer list containing the `MyArray`'s element values. When the compiler sees a statement like

```
MyArray ints1{7};
```

it invokes the constructor that accepts the braced-initializer list of integers, not the single-argument constructor that receives the size.

Using the Overloaded Stream Extraction Operator to Fill a `MyArray`

Next, line 28 prompts the user to enter 17 integers. Line 29 uses the `MyArray` **overloaded stream extraction operator (`>>`)** to read the first seven values into `ints1` and the remaining 10 values into `ints2` (recall that each `MyArray` knows its own size). Line 31 displays each `MyArray`'s updated contents using the **overloaded stream insertion operator (`<<`)**.

```
27    // input and print ints1 and ints2
28    std::cout << "\n\nEnter 17 integers: ";
29    std::cin >> ints1 >> ints2; // uses overloaded >>
30
31    std::cout << "\nints1: " << ints1 << "\nints2: " << ints2;
32
```

```
Enter 17 integers: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

ints1: {1, 2, 3, 4, 5, 6, 7}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

Using the Overloaded Inequality Operator (`!=`)

Line 36 tests `MyArray`'s **overloaded inequality operator (`!=`)** by evaluating the condition

```
ints1!= ints2
```

20 The program output shows that the `MyArray` objects are not equal. Two `MyArray` objects will be equal if they have the same number of elements and their corresponding element values are identical. As you'll see, we define only `MyArray`'s overloaded `==` operator. In C++20, the compiler autogenerates `!=` if you provide an `==` operator for your type. Operator `!=` simply returns the opposite of `==`.

```
33    // use overloaded inequality (!=) operator
34    std::cout << "\n\nEvaluating: ints1 != ints2\n";
35
```

```
36    if (ints1 != ints2) {
37       std::cout << "ints1 and ints2 are not equal\n\n";
38    }
39
```

```
Evaluating: ints1 != ints2
ints1 and ints2 are not equal
```

Initializing a New `MyArray` with a Copy of an Existing `MyArray`

Line 41 instantiates the `MyArray` object `ints3` and initializes it with a copy of `ints1`'s data. This invokes the `MyArray` **copy constructor** to copy `ints1`'s elements into `ints3`. A **copy constructor** is invoked whenever a copy of an object is needed, such as

- passing an object by value to a function,
- returning an object by value from a function or
- initializing an object with a copy of another object of the same class.

Lines 44–45 display `ints3`'s `size` and contents to confirm that `ints3`'s elements were set correctly by the **copy constructor**.

```
40    // create MyArray ints3 by copying ints1
41    MyArray ints3{ints1}; // invokes copy constructor
42
43    // print ints3 size and contents
44    std::cout << fmt::format("\nints3 size: {}\ncontents: ", ints3.size())
45       << ints3;
46
```

```
MyArray copy constructor

ints3 size: 7
contents: {1, 2, 3, 4, 5, 6, 7}
```

11 SE When a class such as `MyArray` contains both a **copy constructor** and a **move constructor**, the compiler chooses the correct one to use based on the context. In line 41, the compiler chose `MyArray`'s **copy constructor** because variable names, like `ints1`, are *lvalues*. As you'll soon see, a **move constructor** receives an ***rvalue* reference**, which is part of C++11's **move semantics**. **An *rvalue* reference** may not refer to an *lvalue*.

The **copy constructor** can also be invoked by writing line 41 as

```
MyArray ints3 = ints1;
```

In an object's definition, the equal sign does *not* indicate assignment. It invokes the single-argument copy constructor, passing as the argument the value to the `=` symbol's right.

Using the Overloaded Copy Assignment Operator (`=`)

Err SE Line 49 assigns `ints2` to `ints1` to test the **overloaded copy assignment operator (`=`)**. Built-in arrays cannot handle this assignment. An array's name is not a modifiable *lvalue*, so assigning to an array's name causes a compilation error. Line 51 displays both objects' contents to confirm that they're now identical. `MyArray ints1` initially held seven integers, but the overloaded operator **resizes** the dynamically allocated built-in array to hold a copy of `ints2`'s 10 elements. **As with copy constructors and move constructors, if a class contains both a copy assignment operator and a**

move assignment operator, the compiler chooses which one to call based on the arguments. In this case, `ints2` is a variable and thus an *lvalue*, so the copy assignment operator is called. Note in the output that line 49 also resulted in calls to the `MyArray` copy constructor and destructor—you'll see why when we present the assignment operator's implementation in Section 11.6.6.

```
47    // use overloaded copy assignment (=) operator
48    std::cout << "\n\nAssigning ints2 to ints1:\n";
49    ints1 = ints2; // note target MyArray is smaller
50
51    std::cout << "\nints1: " << ints1 << "\nints2: " << ints2;
52
```

```
Assigning ints2 to ints1:
MyArray copy assignment operator
MyArray copy constructor
MyArray destructor

ints1: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
ints2: {8, 9, 10, 11, 12, 13, 14, 15, 16, 17}
```

Using the Overloaded Equality Operator (`==`)

Line 56 compares `ints1` and `ints2` with the **overloaded equality operator (`==`)** to confirm they are indeed identical after the assignment in line 49.

```
53    // use overloaded equality (==) operator
54    std::cout << "\n\nEvaluating: ints1 == ints2\n";
55
56    if (ints1 == ints2) {
57       std::cout << "ints1 and ints2 are equal\n\n";
58    }
59
```

```
Evaluating: ints1 == ints2
ints1 and ints2 are equal
```

Using the Overloaded Subscript Operator (`[]`)

Line 61 uses the **overloaded subscript operator (`[]`)** to refer to `ints1[5]`—an *in-range* element of `ints1`. This indexed (subscripted) name is used to get the value stored in `ints1[5]`. Line 65 uses `ints1[5]` on an assignment's left side as a modifiable *lvalue*[12] to assign a new value, `1000`, to element `5` of `ints1`. We'll see that **`operator[]`** returns a reference to use as the modifiable *lvalue* after confirming `5` is a valid index. Line 71 attempts to assign `1000` to `ints1[15]`. **This index is outside `int1`'s bounds, so the overloaded `operator[]` throws an `out_of_range` exception**. Lines 73–75 catch the exception and display its error message by calling the exception's `what` member function.

12. Recall that an *lvalue* can be declared `const`, in which case it would not be modifiable.

```
60    // use overloaded subscript operator to create an rvalue
61    std::cout << fmt::format("ints1[5] is {}\n\n", ints1[5]);
62
63    // use overloaded subscript operator to create an lvalue
```

```
64    std::cout << "Assigning 1000 to ints1[5]\n";
65    ints1[5] = 1000;
66    std::cout << "ints1: " << ints1;
67
68    // attempt to use out-of-range subscript
69    try {
70       std::cout << "\n\nAttempt to assign 1000 to ints1[15]\n";
71       ints1[15] = 1000; // ERROR: subscript out of range
72    }
73    catch (const std::out_of_range& ex) {
74       std::cout << fmt::format("An exception occurred: {}\n", ex.what());
75    }
76
```

```
ints1[5] is 13

Assigning 1000 to ints1[5]
ints1: {8, 9, 10, 11, 12, 1000, 14, 15, 16, 17}

Attempt to assign 1000 to ints1[15]
An exception occurred: Index out of range
```

The **array subscript operator** `[]` is not restricted for use only with arrays. It also can be used, for example, to select elements from other kinds of *container classes* that maintain collections of items, such as `string`s (collections of characters) and `map`s (collections of key–value pairs, which we'll discuss in Chapter 13, Standard Library Containers and Iterators). Also, when **overloaded** `operator[]` functions are defined, *indices are not required to be integers*. In Chapter 13, we discuss the standard library `map` class that allows indices of other types, such as `string`s.

Creating `MyArray ints4` and Initializing It with the `MyArray` Returned By Function `getArrayByValue`

Line 80 initializes `MyArray ints4` with the result of calling function `getArrayByValue` (lines 10–13), which creates a local `MyArray` containing `10`, `20` and `30`, then returns it *by value*. Then, lines 82–83 display the new `MyArray`'s size and contents.

```
77    // initialize ints4 with contents of the MyArray returned by
78    // getArrayByValue; print size and contents
79    std::cout << "\nInitialize ints4 with temporary MyArray object\n";
80    MyArray ints4{getArrayByValue()};
81
82    std::cout << fmt::format("\nints4 size: {}\ncontents: ", ints4.size())
83       << ints4;
84
```

```
Initialize ints4 with temporary MyArray object
MyArray(initializer_list) constructor

ints4 size: 3
contents: {10, 20, 30}
```

Named Return Value Optimization (NRVO)

Recall from function `getArrayByValue`'s definition (lines 10–13) that it creates and initializes a local `MyArray` using the constructor that receives an `initializer_list` of `int` values (line 11). This constructor displays

```
MyArray(initializer_list) constructor
```

each time it's called. Next, `getArrayByValue` returns that local array *by value* (line 12). You might expect that returning an object by value would make a temporary copy of the object for use in the caller. If it did, this would call `MyArray`'s copy constructor to copy the local `MyArray`. You also might expect the local `MyArray` object to go out of scope and have its destructor called as `getArrayByValue` returns to its caller. However, neither the copy constructor nor the destructor displayed lines of output here.

Perf 17 This is due to a compiler performance optimization called **named return value optimization (NRVO)**. When the compiler sees that a local object is constructed, returned from a function by value, then used to initialize an object in the caller, the compiler instead constructs the object directly in the caller where it will be used, eliminating the temporary object and extra constructor and destructor calls mentioned above. This originally was an optional optimization, but C++17 made it required.[13,14]

13. Sy Brand, "Guaranteed Copy Elision Does Not Elide Copies." C++ Team Blog, February 18, 2019. Accessed January 15, 2022. https://devblogs.microsoft.com/cppblog/guaranteed-copy-elision-does-not-elide-copies/.
14. Richard Smith, "Guaranteed Copy Elision Through Simplified Value Categories," September 27, 2015. Accessed January 15, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0135r0.html.

Creating `MyArray ints5` and Initializing It With the *rvalue* Returned By Function `std::move`

A **copy constructor** is called when you initialize one `MyArray` with another that's represented by an *lvalue*. A copy constructor copies its argument's contents. This is similar to a copy-and-paste operation in a text editor—after the operation, you have two copies of the data.

Perf 11 C++ also supports **move semantics**,[15] which help the compiler eliminate the overhead of unnecessarily copying objects. A move is similar to a cut-and-paste operation in a text editor—the data gets *moved* from the cut location to the paste location. A **move constructor** moves into a new object the resources of an object that's no longer needed. Such a constructor receives a C++11 **rvalue reference**, which you'll see is declared with *TypeName*`&&`. *Rvalue* references may refer only to *rvalues*. Typically, these are temporary objects or objects about to be destroyed—called **expiring values** or **xvalues**.

15. Klaus Iglberger, "Back to Basics: Move Semantics," YouTube Video, June 16, 2019. Accessed January 15, 2022. https://www.youtube.com/watch?v=St0MNEU5b0o.

11 Line 88 uses class `MyArray`'s **move constructor** to initialize `MyArray ints5`, then lines 90–91 display the size and contents of the new `MyArray`. The object `ints4` is an *lvalue*—so it cannot be passed directly to `MyArray`'s move constructor. If you no longer need an object's resources, you can **convert it from an *lvalue* to an *rvalue* reference** by passing the object to the C++11 standard library function `std::move` (from header `<utility>`). **This function casts its argument to an *rvalue* reference**,[16] telling the compiler that `ints4`'s contents are no longer needed. So, line 88 forces `MyArray`'s **move constructor** to be called and `ints4`'s contents are *moved* into `ints5`.

16. More specifically, this is called *xvalue* (for expiring value).

```
85    // convert ints4 to an rvalue reference with std::move and
86    // use the result to initialize MyArray ints5
87    std::cout << "\n\nInitialize ints5 with result of std::move(ints4)\n";
88    MyArray ints5{std::move(ints4)}; // invokes move constructor
89
90    std::cout << fmt::format("\nints5 size: {}\ncontents: ", ints5.size())
91       << ints5
92       << fmt::format("\n\nSize of ints4 is now: {}", ints4.size());
93
```

```
Initialize ints5 with result of std::move(ints4)
MyArray move constructor

ints5 size: 3
contents: {10, 20, 30}

Size of ints4 is now: 0
```

It's recommended that you use `std::move` as shown here **only** if you know the source object passed to `std::move` will never be used again. Once an object has been moved, two valid operations can be performed with it:

- destroying it, and
- using it on the left side of an assignment to give it a new value.

In general, you should not call member functions on a moved-from object. *We do so in line 92 only to prove that the* **move constructor** *indeed moved ints4's resources*—the output shows that `ints4`'s size is now `0`.

Assigning `MyArray ints5` to `ints4` with the Move Assignment Operator

Line 96 uses class `MyArray`'s **move assignment operator** to move `ints5`'s contents (`10`, `20` and `30`) back into `ints4`, then lines 98–99 display the size and contents of `ints4`. Line 96 *explicitly converts* the *lvalue* `ints5` to an **rvalue reference** using `std::move`. This indicates that `ints5` no longer needs its resources, so the compiler can *move* them into `ints4`. In this case, the compiler calls `MyArray`'s **move assignment operator**. *For demo purposes only, line 100 outputs `ints5`'s size to show that the move assignment operator indeed moved its resources.* Again, **you should not call member functions on a moved-from object.**

```
94    // move contents of ints5 into ints4
95    std::cout << "\n\nMove ints5 into ints4 via move assignment\n";
96    ints4 = std::move(ints5); // invokes move assignment
97
98    std::cout << fmt::format("\nints4 size: {}\ncontents: ", ints4.size())
99       << ints4
100      << fmt::format("\n\nSize of ints5 is now: {}", ints5.size());
101
```

```
Move ints5 into ints4 via move assignment
MyArray move assignment operator

ints4 size: 3
```

```
contents: {10, 20, 30}

Size of ints5 is now: 0
```

Converting `MyArray ints5` to a `bool` to Test Whether It's Empty

Many programming languages allow you to use a container-class object like a `MyArray` as a condition to determine whether the container has elements. For class `MyArray`, we defined a `bool` conversion operator that returns `true` if the `MyArray` object contains elements (i.e., its `size` is greater than 0) and `false` otherwise. In contexts that require `bool` values, such as control statement conditions, C++ can invoke an object's `bool` conversion operator implicitly. This is known as a **contextual conversion**. Line 103 uses the `MyArray ints5` as a condition, which calls `MyArray`'s `bool` conversion operator. Since we just moved `ints5`'s resources into `ints4`, `ints5` is now empty, and the operator returns `false`. *Once again, you should not call member functions on moved-from objects*—we do so here only to prove that `ints5`'s resources have been moved.

```
102    // check if ints5 is empty by contextually converting it to a bool
103    if (ints5) {
104       std::cout << "\n\nints5 contains elements\n";
105    }
106    else {
107       std::cout << "\n\nints5 is empty\n";
108    }
109
```

```
ints5 is empty
```

Preincrementing Every `ints4` Element with the Overloaded `++` Operator

Some libraries support **"broadcast" operations** that apply the same operation to every element of a data structure. For example, consider the popular high-performance Python programming language library **NumPy**. This library's **`ndarray` (*n*-dimensional array) data structure** overloads many arithmetic operators. They conveniently perform mathematical operations on *every* element of an `ndarray`. In NumPy, the following Python code adds one to every element of the `ndarray` named `numbers`—no iteration statement is required:

```
numbers += 1 # Python does not have a ++ operator
```

We've added a similar capability to class `MyArray`. Line 111 displays `ints4`'s current contents, then line 112 uses `++ints4` to preincrement the `MyArray`, adding one to each element. This expression's result is the updated `MyArray`. We then use `MyArray`'s **overloaded stream insertion operator (`<<`)** to display the contents.

```
110    // add one to every element of ints4 using preincrement
111    std::cout << "\nints4: " << ints4;
112    std::cout << "\npreincrementing ints4: " << ++ints4;
113
```

```
ints4: {10, 20, 30}
preincrementing ints4: {11, 21, 31}
```

Postincrementing Every `ints4` Element with the Overloaded `++` Operator

Line 115 postincrements the entire `MyArray` with the expression `ints4++`. Recall that a postincrement returns its operand's **previous value**, as confirmed by the program's output. The outputs showing that `MyArray`'s **copy constructor** and **destructor** were called are from the postincrement operator's implementation, which we'll discuss in Section 11.6.14.

```
114    // add one to every element of ints4 using postincrement
115    std::cout << "\n\npostincrementing ints4: " << ints4++ << "\n";
116    std::cout << "\nints4 now contains: " << ints4;
117
```

```
postincrementing ints4: MyArray copy constructor
{11, 21, 31}
MyArray destructor

ints4 now contains: {12, 22, 32}
```

Adding a Value to Every `ints4` Element with the Overloaded `+=` Operator

Class `MyArray` also provides a broadcast-based **overloaded addition assignment operator (`+=`)** for adding an `int` value to every `MyArray` element. Line 119 adds 7 to every `ints4` element, then displays its new contents. Note that the non-overloaded versions of `++` and `+=` still work on individual `MyArray` elements, too—those are simply `int` values.

```
118       // add a value to every element of ints4 using +=
119       std::cout << "\n\nAdd 7 to every ints4 element: " << (ints4 += 7)
120        << "\n";
121    }
```

```
Add 7 to every ints4 element: {19, 29, 39}
```

Destroying the `MyArray` Objects That Remain

When function `main` terminates, the **destructors** are called for the five `MyArray` objects created in `main`, producing the last five lines of the program's output.

```
MyArray destructor
MyArray destructor
MyArray destructor
MyArray destructor
MyArray destructor
```

11.6.3 `MyArray` Class Definition

Now, let's walk through `MyArray`'s header (Fig. 11.4). As we refer to each member function in the header, we discuss that function's implementation in Fig. 11.5. We've broken the member-function implementation file into small segments for discussion purposes.

```
1    // Fig. 11.4: MyArray.h
2    // MyArray class definition with overloaded operators.
```

```
3   #pragma once
4   #include <initializer_list>
5   #include <iostream>
6   #include <memory>
7
8   class MyArray final {
9      // overloaded stream extraction operator
10     friend std::istream& operator>>(std::istream& in, MyArray& a);
11
12     // used by copy assignment operator to implement copy-and-swap idiom
13     friend void swap(MyArray& a, MyArray& b) noexcept;
14
15  public:
16     explicit MyArray(size_t size); // construct a MyArray of size elements
17
18      // construct a MyArray with a braced-initializer list of ints
19     explicit MyArray(std::initializer_list<int> list);
20
21     MyArray(const MyArray& original); // copy constructor
22     MyArray& operator=(const MyArray& right); // copy assignment operator
23
24     MyArray(MyArray&& original) noexcept; // move constructor
25     MyArray& operator=(MyArray&& right) noexcept; // move assignment
26
27     ~MyArray(); // destructor
28
29     size_t size() const noexcept {return m_size;}; // return size
30     std::string toString() const; // create string representation
31
32     // equality operator
33     bool operator==(const MyArray& right) const noexcept;
34
35     // subscript operator for non-const objects returns modifiable lvalue
36     int& operator[](size_t index);
37
38     // subscript operator for const objects returns non-modifiable lvalue
39     const int& operator[](size_t index) const;
40
41     // convert MyArray to a bool value: true if non-empty; false if empty
42     explicit operator bool() const noexcept {return size() != 0;}
43
44     // preincrement every element, then return updated MyArray
45     MyArray& operator++();
46
47     // postincrement every element, and return copy of original MyArray
48     MyArray operator++(int);
49
50     // add value to every element, then return updated MyArray
51     MyArray& operator+=(int value);
52  private:
53     size_t m_size{0}; // pointer-based array size
54     std::unique_ptr<int[]> m_ptr; // smart pointer to integer array
55  };
56
57  // overloaded operator<< is not a friend--does not access private data
```

```
58    std::ostream& operator<<(std::ostream& out, const MyArray& a);
```

Fig. 11.4 | `MyArray` class definition with overloaded operators.

In Fig. 11.4, lines 53–54 declare `MyArray`'s `private` data members:

- `m_size` stores its number of elements.
- `m_ptr` is a `unique_ptr` that manages a dynamically allocated pointer-based `int` array containing the `MyArray` object's elements. When a `MyArray` goes out of scope, its destructor will call the `unique_ptr`'s destructor, automatically deleting the dynamically allocated memory.

Throughout this class's member-function implementations, we use some of C++'s declarative, functional-style programming capabilities discussed in Chapters 6 and 7. We also introduce three additional standard library algorithms—`copy`, `for_each` and `equal`.

11.6.4 Constructor That Specifies a `MyArray`'s Size

Line 16 of Fig. 11.4

```
explicit MyArray(size_t size); // construct a MyArray of size elements
```

declares a **constructor** that specifies the number of `MyArray` elements. Theconstructor's definition (Fig. 11.5, lines 15–19) performs several tasks:

- Line 16 initializes the `m_size` member using the argument `size`.
- Line 17 initializes the `m_ptr` member to a `unique_ptr` returned by the standard library's `make_unique` function template (see Section 11.5). Here, we use it to create a dynamically allocated `int` array of `size` elements. The function `make_unique` **value initializes the dynamic memory** it allocates, so the `int` array's elements are set to 0.
- For pedagogic purposes, line 18 displays that the constructor was called. We do this in all of `MyArray`'s **special member functions and other constructors** to give you visual confirmation that the functions are being called.

```
 1  // Fig. 11.5: MyArray.cpp
 2  // MyArray class member- and friend-function definitions.
 3  #include <algorithm>
 4  #include <fmt/format.h>
 5  #include <initializer_list>
 6  #include <iostream>
 7  #include <memory>
 8  #include <span>
 9  #include <sstream>
10  #include <stdexcept>
11  #include <utility>
12  #include "MyArray.h" // MyArray class definition
13
14  // MyArray constructor to create a MyArray of size elements containing 0
15  MyArray::MyArray(size_t size)
16     : m_size{size},
17       m_ptr{std::make_unique<int[]>(size)} {
18     std::cout << "MyArray(size_t) constructor\n";
19  }
```

```
20
```

Fig. 11.5 `MyArray` class member- and friend-function definitions.

11 11.6.5 C++11 Passing a Braced Initializer to a Constructor

In Fig. 6.2, we initialized a `std::array` object with a braced-initializer list, as in

```
std::array<int, 5> n{32, 27, 64, 18, 95};
```

You can use **braced initializers** for objects of your own classes by providing a **constructor** with a **`std::initializer_list`** parameter, as declared in line 19 of Fig. 11.4:

```
explicit MyArray(std::initializer_list<int> list);
```

The `std::initializer_list` class template is defined in **`<initializer_list>`**. With this constructor, we can create `MyArray` objects that initialize their elements as they're constructed, as in

```
MyArray ints{10, 20, 30};
```

or

```
MyArray ints = {10, 20, 30};
```

SE Each creates a three-element `MyArray` containing `10`, `20` and `30`. **In a class that provides an `initializer_list` constructor, the class's other single-argument constructors must be called using parentheses rather than braces.** The **braced-initialization constructor**(lines 22–29) has one `initializer_list<int>` parameter named `list`. You can determine `list`'s number of elements by calling its **`size` member function** (line 23).

```
21  // MyArray constructor that accepts an initializer list
22  MyArray::MyArray(std::initializer_list<int> list)
23     : m_size{list.size()}, m_ptr{std::make_unique<int[]>(list.size())} {
24     std::cout << "MyArray(initializer_list) constructor\n";
25
26     // copy list argument's elements into m_ptr's underlying int array
27     // m_ptr.get() returns the int array's starting memory location
28     std::copy(std::begin(list), std::end(list), m_ptr.get());
29  }
30
```

To copy each initializer list value into the new `MyArray` object, line 28 uses the standard library **`copy`** algorithm (from header **`<algorithm>`**) to copy each `initializer_list` element into the new `MyArray`. The algorithm copies each element in the range specified by its first two arguments—the beginning and end of the `initializer_list`. These elements are copied into the destination specified by `copy`'s third argument. The `unique_ptr`'s **`get` member function** returns the `int*` that points to the first element of the `MyArray`'s underlying `int` array.

CG Sections 9.15 and 9.17 introduced the **compiler-generated default copy assignment operator** and **default copy constructor**. These performed **memberwise copy operations by default**. The C++ Core Guidelines recommend designing your classes such that the compiler can autogenerate the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor. This is known as the **Rule of Zero**.[17] You can accomplish this by composing each class's data using fundamental-type members and objects of classes that do not require you to implement custom resource processing, such as standard library classes like `array` and `vector`, which each use RAII to manage resources.

17. C++ Core Guidelines, "C.20: If You Can Avoid Defining Any Default Operations, Do." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-zero.

The Rule of Five

CG The default special member functions work well for fundamental-type values like `int`s and `double`s. But what about objects of types that manage their own resources, such as pointers to dynamically allocated memory? Classes that manage their own resources should define the five special member functions. The C++ Core Guidelines state that if a class requires one special member function, it should define them all,[18] as we do in this case study. This is known as the **Rule of Five**.

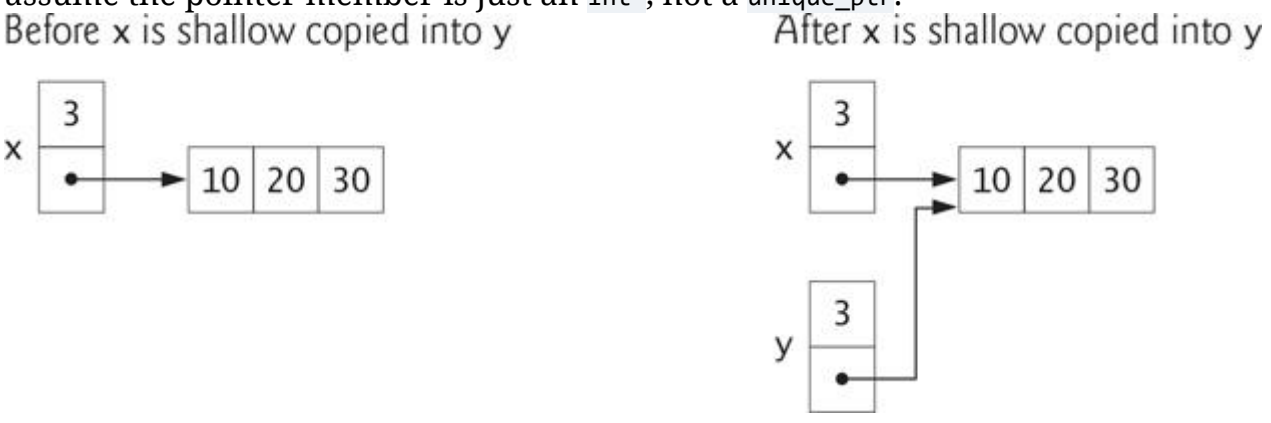
18. C++ Core Guidelines, "C.21: If You Define or `=delete` Any Copy, Move, or Destructor Function, Define or `=delete` Them All." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-five.

11 Even for classes with the compiler-generated special member functions, some experts recommend declaring them explicitly in the class definition with `= default` (introduced in Chapter 10). This is called the **Rule of Five `defaults`**.[19] You also can explicitly remove compiler-generated special member functions to prevent the specified functionality by following their function prototypes with C++11's **`= delete`**. Class `unique_ptr` actually does this for the copy constructor and copy assignment operator.

19. Scott Meyers, "A Concern About the Rule of Zero," March 13, 2014. Accessed January 15, 2022. http://scottmeyers.blogspot.com/2014/03/a-concern-about-rule-of-zero.html.

Shallow Copy

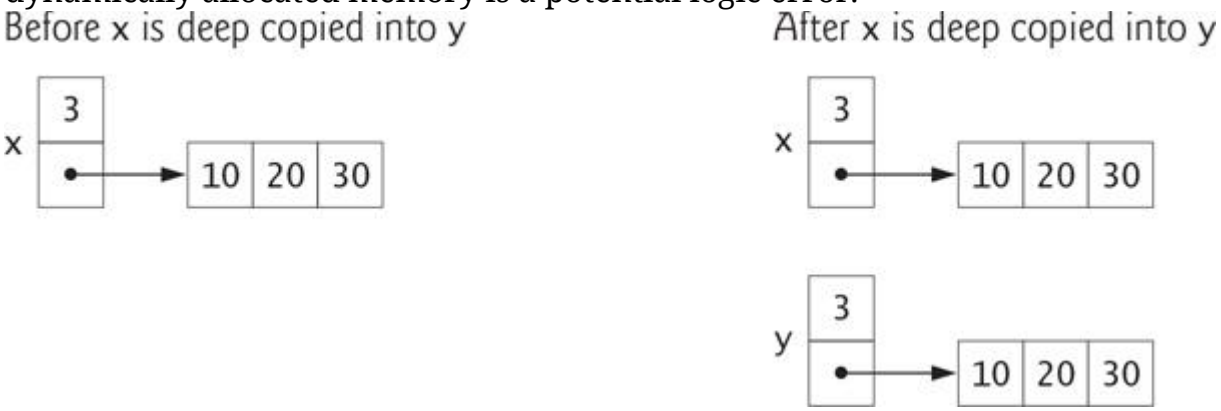
The compiler-generated copy constructor and copy assignment operator perform memberwise **shallow copies**. If the member is a pointer to dynamically allocated memory, only the address in the pointer is copied. In the following diagram, consider the object `x`, which contains its number of elements (`3`) and a pointer to a dynamically allocated array. For this discussion, let's assume the pointer member is just an `int*`, not a `unique_ptr`.



Err Now, assume we'd like to copy `x` into a new object `y`. If the **copy constructor** simply copied the pointer in `x` into the target object `y`'s pointer, both would point to the same **dynamically allocated memory**, as in the right side of the diagram. **The first destructor to execute would delete the memory that is shared between these two objects. The other object's pointer would then point to memory that's no longer allocated.** This situation is called a **dangling pointer**. Typically, this would result in a serious runtime error (such as early program termination) if the program were to dereference that pointer—accessing deleted memory is undefined behavior.

Deep Copy

Err **In classes that manage their own resources, copying must be done carefully to avoid the pitfalls of shallow copy.** Classes that manage their objects' resources should define their own **copy constructor** and **overloaded copy assignment operator** to perform **deep copies**. The following diagram shows that after the object `x` is deep copied into the object `y`, both objects have their own copies of the dynamically allocated array containing `10`, `20` and `30`. Not providing a copy constructor and overloaded assignment operator for a class when objects of that class contain pointers to dynamically allocated memory is a potential logic error.



Implementing the Copy Constructor

Line 21 of Fig. 11.4

```
MyArray(const MyArray& original); // copy constructor
```

SE declares the class's **copy constructor** (defined lines 32–41). Its argument must be a **`const` reference** to prevent the constructor from modifying the argument object's data.

```
31  // copy constructor: must receive a reference to a MyArray
32  MyArray::MyArray(const MyArray& original)
33     : m_size{original.size()},
34       m_ptr{std::make_unique<int[]>(original.size())} {
35     std::cout << "MyArray copy constructor\n";
36
37     // copy original's elements into m_ptr's underlying int array
38     const std::span<const int> source{
39        original.m_ptr.get(), original.size()};
40     std::copy(std::begin(source), std::end(source), m_ptr.get());
```

```
41   }
42
```

When the **copy constructor** is called to initialize a new `MyArray` by copying an existing one, it performs the following tasks:

- Line 33 initializes the `m_size` member using the return value of `original`'s `size` member function.
- Line 34 initializes the `m_ptr` member to a `unique_ptr` returned by the standard library's `make_unique` function template, which creates a dynamically allocated `int` array containing `original.size()` elements.
- Line 35 outputs that the **copy constructor** was called.
- Recall that a `span` is a view into a contiguous collection of items, such as an array. Lines 38–39 create a **`span`** named `source` representing the argument `MyArray`'s **dynamically allocated `int` array** from which we'll copy elements.
- Line 40 copies the elements in the range represented by the beginning and end of the `source` `span` into the `MyArray`'s underlying `int` array.

Copy Assignment Operator (=)

Line 22 of Fig. 11.4

```
MyArray& operator=(const MyArray& right); // copy assignment operator
```

declares the class's **overloaded copy assignment operator (=)**.[20] This function's definition (lines 44–49) enables one `MyArray` to be assigned to another, copying the contents from the right operand into the left. When the compiler sees the statement

20. This copy assignment operator ensures that the `MyArray` object is not modified and no memory resources are leaked if an exception occurs. This is known as a strong exception guarantee (see Section 12.3).

```
ints1 = ints2;
```

it invokes member function `operator=` with the call

```
ints1.operator=(ints2)
```

```
43   // copy assignment operator: implemented with copy-and-swap idiom
44   MyArray& MyArray::operator=(const MyArray& right) {
45      std::cout << "MyArray copy assignment operator\n";
46      MyArray temp{right}; // invoke copy constructor
47      swap(*this, temp); // exchange contents of this object and temp
48      return *this;
49   }
50
```

We could implement the **overloaded copy assignment operator** similarly to the **copy constructor**. However, there's an elegant way to use the **copy constructor** to implement the **overloaded copy assignment operator**—the copy-and-swap idiom.[21,22] The idiom operates as follows:

21. Herb Sutter, “Exception-Safe Class Design, Part 1: Copy Assignment.” Accessed January 15, 2022. http://www.gotw.ca/gotw/059.htm.
22. Answer to “What is the copy-and-Swap Idiom?” Edited April 24, 2021, by Jack Lilhammers. Accessed January 15, 2022. https://stackoverflow.com/a/3279550.

- First, it copies the argument `right` into a local `MyArray` object (`temp`) using the **copy constructor** (line 46). If this fails to allocate memory for `temp`’s array, a `bad_alloc` **exception** will occur. In this case, the **overloaded copy assignment operator** will terminate without modifying the object on the assignment’s left.
- Line 47 uses class `MyArray`’s `friend` function `swap` (defined in lines 169–172) to exchange the contents of `*this` (the object on the assignment’s left) with `temp`.
- Finally, the function returns a reference to the current object (`*this` in line 48), enabling cascaded `MyArray` assignments such as `x = y = z`.

When the function returns to its caller, the `temp` object’s **destructor** is called to **release the memory** managed by the `temp` object’s `unique_ptr`. Line 46’s copy constructor call and the destructor call when `temp` goes out of scope are the reason for the two additional lines of output you saw when we demonstrated assigning `ints2` to `ints1` in line 49 of Fig. 11.3.

11.6.7 Move Constructor and Move Assignment Operator

Perf Often an object being copied is about to be destroyed, such as a local object returned from a function by value. It’s more efficient to move that object’s contents into the destination object to eliminate the copying overhead. That’s the purpose of the **move constructor** and **move assignment operator** declared in lines 24–25 of Fig. 11.4:

```
MyArray(MyArray&& original) noexcept; // move constructor
MyArray& operator=(MyArray&& right) noexcept; // move assignment
```

Each receives an ***rvalue* reference** declared with `&&` to distinguish it from an *lvalue* reference `&`. ***Rvalue* references** help implement **move semantics**. Rather than copying the argument, the **move constructor** and **move assignment operator** each move their argument object’s data, leaving the original object in a state that can be destructed properly.

`noexcept` Specifier

11 Err As of C++11, if a function does not throw any exceptions *and* does not call any functions that throw exceptions, you should explicitly state that the function does not throw exceptions.[23] Simply add `noexcept` after the function’s signature in both the prototype and the definition. For a `const` member function, keyword `noexcept` must be placed after `const`. If a `noexcept` function calls another function that throws an exception and the `noexcept` function does not handle that exception, the program terminates immediately.
23. C++ Core Guidelines, “F.6: If Your Function May Not Throw, Declare It `noexcept`.” Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-noexcept.

The `noexcept` specification can optionally be followed by parentheses containing a `bool` expression that evaluates to `true` or `false`. `noexcept` by itself is equivalent to `noexcept(true)`. Following a function’s signature with `noexcept(false)` indicates that the class’s designer has thought about whether the function might throw exceptions and has decided it might. In such cases, client-code programmers can decide whether to wrap calls to the function in `try` statements.

Class `MyArray`'s Move Constructor

The **move constructor** (lines 52–56) declares its parameter as an ***rvalue* reference** (**&&**) to indicate that its `MyArray` argument must be a temporary object. The member-initializer list moves members `m_size` and `m_ptr` from the argument into the object being constructed.

```
51  // move constructor: must receive an rvalue reference to a MyArray
52  MyArray::MyArray(MyArray&& original) noexcept
53     : m_size{std::exchange(original.m_size, 0)},
54       m_ptr{std::move(original.m_ptr)} { // move original.m_ptr into m_ptr
55     std::cout << "MyArray move constructor\n";
56  }
57
```

Recall from Section 11.6.2 that the only valid operations on a moved-from object are assigning another object to it or destroying it. **When moving resources from an object, the object should be left in a state that allows it to be properly destructed. Also, it should no longer refer to the resources that were moved to the new object.**[24] To accomplish this for the `m_size` member, line 53

24. C++ Core Guidelines, "C.64: A Move Operation Should Move and Leave Its Source in a Valid State." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-semantic.

- calls the standard library `exchange` function (header `<utility>`), which sets its first argument (`original.m_size`) to its second argument's value (`0`) and returns its first argument's original value, then
- initializes the new object's `m_size` with the value that exchange returns.

When a `unique_ptr` is **move constructed**, as in line 54, its **move constructor** transfers ownership of the source `unique_ptr`'s dynamic memory to the target `unique_ptr` and sets the source `unique_ptr` to `nullptr`.[25] If your class manages raw pointers, you'd have to explicitly set the source pointer to `nullptr`—or use `exchange`, similar to line 53.

25. "`std::unique_ptr<T,Deleter>::unique_ptr`." Accessed January 15, 2022. https://en.cppreference.com/w/cpp/memory/unique_ptr/unique_ptr.

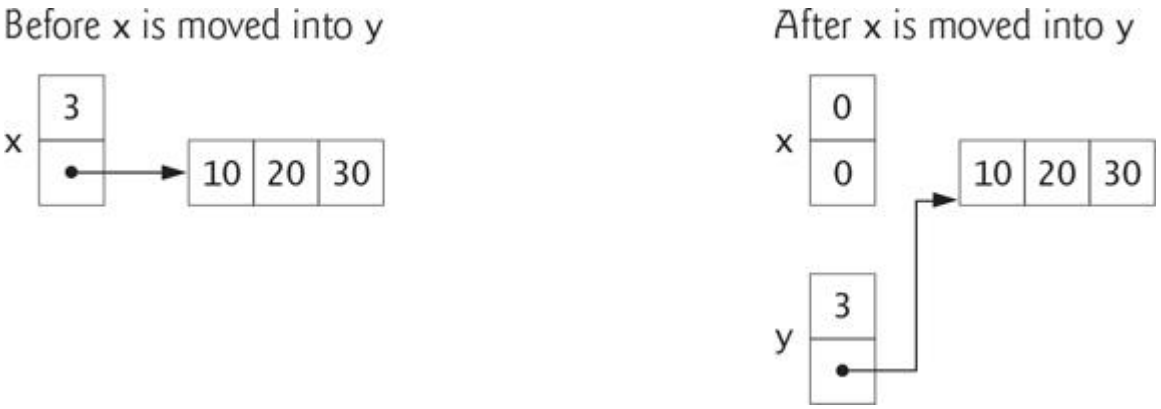
Moving Does Not Move Anything

Though we said the move constructor "moves the `m_size` and `m_ptr` members from the argument object into the object being constructed," it does not actually move anything:[26]

26. Topher Winward, "C++ Moves For People Who Don't Know or Care What Rvalues Are," January 17, 2019. Accessed January 15, 2022. https://medium.com/@winwardo/c-moves-for-people-who-dont-know-or-care-what-rvalues-are-%EF%B8%8F-56ee122dda7.

- For fundamental types like `size_t`, which is simply an unsigned integer, **the value is copied** from the source object's member to the new object's member.
- For a raw pointer, the address stored in the source object's pointer is copied to the new object's pointer.
- For an object, the object's move constructor is called. A `unique_ptr`'s move constructor **transfers ownership** of the dynamic memory by copying the dynamically allocated memory's address from the source `unique_ptr`'s underlying raw pointer into the new object's underlying raw pointer, then assigning `nullptr` to the source `unique_ptr` to indicate it no longer manages any data.

The diagram below shows the concept of moving a source object `x` with a raw pointer member into a new object `y`. Note that both members of `x` are `0` after the move—`0` in a pointer member represents a null pointer.



Class `MyArray`'s Move Assignment Operator (`=`)

The **move assignment operator** (`=`) (lines 59–69) defines an ***rvalue* reference** (`&&`) parameter to indicate that its `MyArray` argument's resources should be moved (not copied). Line 62 tests for **self-assignment** in which a `MyArray` object is being **assigned to itself.**[27] When `this` is equal to the `right` operand's address, the same object is on both sides of the assignment, so there's no need to move anything.

27. "C.65: Make Move Assignment Safe for Self-Assignment." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-self.

```
58  // move assignment operator
59  MyArray& MyArray::operator=(MyArray&& right) noexcept {
60     std::cout << "MyArray move assignment operator\n";
61
62     if (this != &right) { // avoid self-assignment
63        // move right's data into this MyArray
64        m_size = std::exchange(right.m_size, 0); // indicate right is empty
65        m_ptr = std::move(right.m_ptr);
66     }
67
68     return *this; // enables x = y = z, for example
69  }
70
```

If it is not a **self-assignment**, lines 64–65

- move `right.m_size` into the target `MyArray`'s `m_size` by calling `exchange`—this sets `right.m_size` to `0` and returns its original value, which we use to set the target `MyArray`'s `m_size` member, and
- move `right.m_ptr` into the target `MyArray`'s `m_ptr`.

As in the **move constructor**, when a `unique_ptr` is **move assigned**, ownership of its dynamic memory transfers to the new `unique_ptr`, and the **move assignment operator** sets the original `unique_ptr` to `nullptr`. Regardless of whether this is a self-assignment, the member function returns the current object (`*this`), which enables cascaded `MyArray` assignments such as `x = y = z`.

11 Move Operations Should Be `noexcept`

SE CG **Move constructors and move assignment operators should never throw exceptions.** They do not acquire any new resources—they simply *move* existing ones. For this reason, the C++ Core Guidelines recommend declaring **move constructors** and **move assignment**

operators `noexcept`.[28] This also is a requirement to be able to use your class's move capabilities with the standard library's containers like `vector`.

28. C++ Core Guidelines, "C.66: Make Move Operations `noexcept`." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-move-noexcept.

11.6.8 Destructor

Line 27 of Fig. 11.4

```
~MyArray(); // destructor
```

CG declares the class's **destructor** (defined in lines 73–75), which is invoked when a `MyArray` object goes out of scope. This will automatically call `m_ptr`'s destructor, which will **release the dynamically allocated `int` array created when the `MyArray` object was constructed. The C++ Core Guidelines indicate that it's poor design if destructors throw exceptions. For this reason, they recommend declaring destructors `noexcept`.**[29] However, unless your class is derived from a base class with a destructor that's declared `noexcept(false)`, the compiler implicitly declares the destructor `noexcept` by default.

29. C++ Core Guidelines, "C.37: Make Destructors `noexcept`." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-noexcept.

```
71  // destructor: This could be compiler-generated. We included it here so
72  // we could output when each MyArray is destroyed.
73  MyArray::~MyArray() {
74     std::cout << "MyArray destructor\n";
75  }
76
```

11.6.9 `toString` and `size` Functions

Line 29 in Fig. 11.4

```
size_t size() const noexcept {return m_size;}; // return size
```

defines an inline `size` member function, which returns a `MyArray`'s number of elements.

Line 30 in Fig. 11.4s

```
std::string toString() const; // create string representation
```

declares a `toString` member function (defined in lines 78–91), which returns the `string` representation of a `MyArray`'s contents. Function `toString` uses an `ostringstream` (introduced in Section 8.17) to build a `string` containing the `MyArray`'s element values enclosed in braces (`{}`) and separating each `int` from the next by a comma and a space.

```
77  // return a string representation of a MyArray
78  std::string MyArray::toString() const {
79     const std::span<const int> items{m_ptr.get(), m_size};
80     std::ostringstream output;
81     output << "{";
82
83     // insert each item in the dynamic array into the ostringstream
84     for (size_t count{0}; const auto& item : items) {
85        ++count;
86        output << item << (count < m_size ? ", " : "");
87     }
88
89     output << "}";
90     return output.str();
91  }
92
```

11.6.10 Overloading the Equality (==) and Inequality (!=) Operators

Line 33 of Fig. 11.4

```
bool operator==(const MyArray& right) const noexcept;
```

declares the **overloaded equality operator (==)**. Comparisons should not throw exceptions, so they should be declared `noexcept`.[30]

30. C++ Core Guidelines, "C.86: Make == Symmetric with Respect Of Operand Types and `noexcept`." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq.

When the compiler sees an expression like `ints1 == ints2`, it invokes this overloaded operator with the call

```
ints1.operator==(ints2)
```

Member function `operator==` (defined in lines 95–101) operates as follows:

- Line 97 creates the `span lhs` representing the dynamically allocated `int` array in the left-hand-side operand (`ints1`).
- Line 98 creates the `span rhs` representing the dynamically allocated `int` array in the right-hand-side operand (`ints2`).
- Lines 99–100 use the standard library algorithm `equal` (from header `<algorithm>`) to compare corresponding elements from each `span`. The first two arguments specify the `lhs` object's range of elements. The last two specify the `rhs` object's range of elements. If the `lhs` and `rhs` objects have different lengths or if any pair of corresponding elements differ, `equal` returns `false`. If every pair of elements is equal, `equal` returns `true`.

```
93  // determine if two MyArrays are equal and
94  // return true, otherwise return false
```

```
95  bool MyArray::operator==(const MyArray& right) const noexcept {
96     // compare corresponding elements of both MyArrays
97     const std::span<const int> lhs{m_ptr.get(), size()};
98     const std::span<const int> rhs{right.m_ptr.get(), right.size()};
99     return std::equal(std::begin(lhs), std::end(lhs),
100       std::begin(rhs), std::end(rhs));
101 }
102
```

Compiler Generates the `!=` Operator

20 As of C++20, *the compiler autogenerates a `!=` operator function for you if you provide the `==` operator for your type.* Prior to C++20, if your class required a custom **overloaded inequality operator (`!=`)** operator, you'd typically define `!=` to call the `==` operator function and return the opposite result.

Defining Comparison Operators as Non-Member Functions

CG CG Each class we've defined so far, including `MyArray`, declared its single-argument constructor(s) `explicit` to prevent implicit conversions, as recommended by the C++ Core Guideline "C.164: Avoid Implicit Conversion Operators."[31] You may also come across the C++ Core Guideline "C.86: Make `==` Symmetric with Respect to Operand Types and `noexcept`,"[32] which recommends making comparison operators non-member functions **if your class supports implicitly converting objects of other types to your class's type, or vice versa**. This guideline applies to all comparison operators, but we'll discuss `==` here, as it's the only comparison operator defined in our `MyArray` class.

31. C++ Core Guidelines, "C.164: Avoid Implicit Conversion Operators," Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ro-conversion.

32. C++ Core Guidelines, "C.86: Make `==` Symmetric with Respect to Operand Types and `noexcept`." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-eq.

Assume `ints` is a `MyArray` and `other` is an object of class `OtherType`, which is **implicitly convertible to a `MyArray`**. To satisfy Core Guideline C.86, we'd define `operator==` as a non-member function with the prototype

```
bool operator==(const MyArray& left, const MyArray& right) noexcept;
```

This would allow the following mixed-type expressions:

```
ints == other
```

or

```
other == ints
```

SE There is no `operator==` definition that provides parameters exactly matching operands of types `MyArray` and `OtherType` or `OtherType` and `MyArray`. However, **C++ allows one user-defined conversion per expression**. So, if `OtherType` objects can be implicitly converted to `MyArray` objects, the compiler will convert `other` to a `MyArray`, then call the `operator==` that receives two `MyArray`s.

C++ follows a complex set of overload-resolution rules to determine which function to call for each operator expression.[33] If `operator==` is a `MyArray` member function, **the *left*** operand must be a `MyArray`. C++ will not implicitly convert `other` to a `MyArray` to call the member function, so the expression

33. "Overload Resolution." Accessed January 15, 2022. https://en.cppreference.com/w/cpp/language/overload_resolution.

```
other == ints
```

with an `OtherType` object on the left causes a compilation error. This might confuse programmers who'd expect this expression to compile, which is why Core Guideline C.86 recommends using a non-member `operator==` function.

11.6.11 Overloading the Subscript ([]) Operator

Lines 36 and 39 of Fig. 11.4

```
int& operator[](size_t index);
cont int& operator[](size_t index) const;
```

declare **overloaded subscript operators** (defined in lines 105–112 and 116–123). When the compiler sees an expression like `ints1[5]`, it invokes the appropriate **overloaded `operator[]` member function** by generating the call

```
ints1.operator[](5)
```

The compiler calls the `const` version of **`operator[]`** (lines 116–123) when the **subscript operator** is used on a `const MyArray` object. For example, if you pass a `MyArray` to a function that receives the `MyArray` as a `const MyArray&` named `z`, then the **`const` version of `operator[]`** is required to execute a statement such as

```
std::cout << z[3];
```

Remember that when an object is `const`, a program can invoke only the object's `const`

```
103  // overloaded subscript operator for non-const MyArrays;
104  // reference return creates a modifiable lvalue
105  int& MyArray::operator[](size_t index) {
106     // check for index out-of-range error
107     if (index >= m_size) {
108        throw std::out_of_range{"Index out of range"};
109     }
110
111     return m_ptr[index]; // reference return
112  }
113
114  // overloaded subscript operator for const MyArrays
115  // const reference return creates a non-modifiable lvalue
```

```
116  const int& MyArray::operator[](size_t index) const {
117     // check for subscript out-of-range error
118     if (index >= m_size) {
119        throw std::out_of_range{"Index out of range"};
120     }
121
122     return m_ptr[index]; // returns copy of this element
123 }
124
```

Each **`operator[]`** definition determines whether argument `index` is in range. If not, it throws an **`out_of_range` exception** (header `<stdexcept>`). If `index` is in range, the **non-`const` version of `operator[]`** returns the appropriate `MyArray` element as a reference. This may be used as a modifiable *lvalue* on an assignment's left side to modify an array element. The **`const` version of `operator[]`** returns a `const` reference to the appropriate array element.

The subscript operators used in lines 111 and 122 belong to class `unique_ptr`. When a `unique_ptr` manages a dynamically allocated array, `unique_ptr`'s overloaded `[]` operator enables you to access the array's elements.

11.6.12 Overloading the Unary `bool` Conversion Operator

You can define your own **conversion operators** for converting between types—these are also called **overloaded cast operators**. Line 42 of Fig. 11.4

```
explicit operator bool() const noexcept {return size() != 0;}
```

defines an **inline** overloaded `operator bool` that converts a `MyArray` object to the `bool` value `true` if the `MyArray` is not empty or `false` if it is. Like single-argument constructors, we declared this operator `explicit` to prevent the compiler from using it for implicit conversions (we say more about this in Section 11.9). **Overloaded conversion operators do not specify a return type to the left of the `operator` keyword. The return type is the conversion operator's type—`bool` in this case.**

In Fig. 11.3, line 103 used the `MyArray ints5` as an `if` statement's condition to determine whether it contained elements. In that case, **C++ called this `operator bool` function to perform a contextual conversion of the `ints5` object to a `bool` value for use as a condition.** You also may call this function explicitly using an expression like:

```
static_cast<bool>(ints5)
```

We say more about converting between types in Section 11.8.

11.6.13 Overloading the Preincrement Operator

You can overload the prefix and postfix increment and decrement operators. The concepts we show here and in Section 11.6.14 for `++` also apply to the `--` operators. Section 11.6.14 shows how the compiler distinguishes between the prefix and postfix versions.

Line 45 of Fig. 11.4

```
MyArray& operator++();
```

declares `MyArray`'s unary **overloaded preincrement operator (`++`)**. When the compiler sees an expression like `++ints4`, it invokes `MyArray`'s overloaded preincrement operator (`++`) function by generating the call

```
ints4.operator++()
```

This invokes the function in lines 126–132, which adds one to each element by

- creating the `span items` to represent the dynamically allocated `int` array then
- using the standard library's `for_each` algorithm to call a function that performs a task once for each element of the `span`.

```
125  // preincrement every element, then return updated MyArray
126  MyArray& MyArray::operator++() {
127     // use a span and for_each to increment every element
128     const std::span<int> items{m_ptr.get(), m_size};
129     std::for_each(std::begin(items), std::end(items),
130        [](auto& item){++item;});
131     return *this;
132  }
133
```

Like the `copy` algorithm, `for_each`'s first two arguments represent the range of elements to process. The third argument is a function that receives one argument and performs a task with it. In this case, we specify a **lambda expression** that is called once for each element in the range. As `for_each` iterates internally through the `span`'s elements, it passes the current element as the lambda's argument (`item`). The lambda then performs a task using that value. This lambda's argument is a non-`const` reference (`auto&`), so the expression `++item` in the lambda's body modifies the original element in the `MyArray`.

The operator returns a reference to the `MyArray` object it just incremented. This enables a preincremented `MyArray` object to be used as an *lvalue*, which is how the built-in prefix increment operator works for fundamental types.

11.6.14 Overloading the Postincrement Operator

SE Overloading the postfix increment operator presents a challenge. The compiler must be able to distinguish between the signatures of the overloaded prefix and postfix increment operator functions. By convention, when the compiler sees a postincrement expression like `ints4++`, it generates the member-function call

```
ints4.operator++(0)
```

The argument `0` is strictly a **dummy value** that the compiler uses to distinguish between the prefix and postfix increment operator functions. The same syntax differentiates the prefix and postfix decrement operator functions.

Line 48 of Fig. 11.4

```
MyArray operator++(int);
```

declares `MyArray`'s unary **overloaded postincrement operator (`++`)** with the `int` parameter that receives the dummy value `0`. The parameter is not used, so it's declared without a parameter name. To emulate the effect of the postincrement, we must return an **unincremented copy** of the `MyArray` object. So, the function's definition (lines 135–139)

- uses the `MyArray` **copy constructor** to make a local copy of the original `MyArray`,
- calls the preincrement operator to add one to every element of the `MyArray`[34], then

34. Herb Sutter, "GotW #2 Solution: Temporary Objects," May 13, 2013. Accessed January 15, 2022. https://herbsutter.com/2013/05/13/gotw-2-solution-temporary-objects/.

- Perf returns the unincremented local copy of the `MyArray` *by value*—this is another case in which compilers can use the **named return value optimization (NRVO)**.

Perf The extra local object created by the postfix increment (or decrement) operator can result in a performance problem, especially if the operator is used in a loop. For this reason, you should **prefer the prefix increment and decrement operators.**

```
134  // postincrement every element, and return copy of original MyArray
135  MyArray MyArray::operator++(int) {
136     MyArray temp(*this);
137     ++(*this); // call preincrement operator++ to do the incrementing
138     return temp; // return the temporary copy made before incrementing
139  }
140
```

11.6.15 Overloading the Addition Assignment Operator (`+=`)

Line 51 of Fig. 11.4

```
MyArray& operator+=(int value);
```

declares `MyArray`'s **overloaded addition assignment operator (`+=`)**, which adds a value to every element of a `MyArray`, then **returns a reference to the modified object** to enable cascaded calls. Like the preincrement operator, we use a `span` and the standard library function `for_each` to process every element in the `MyArray`. In this case, the lambda we pass as `for_each`'s last argument (line 146) uses the `operator+=` function's `value` parameter in its body. The **lambda introducer** `[value]` specifies that the compiler should allow `value` to be used in the lambda's body—this is known as **capturing** the variable. We'll say more about capturing lambdas in Chapter 14.

```
141  // add value to every element, then return updated MyArray
142  MyArray& MyArray::operator+=(int value) {
143     // use a span and for_each to increment every element
144     const std::span<int> items{m_ptr.get(), m_size};
145     std::for_each(std::begin(items), std::end(items),
146        [value](auto& item) {item += value;});
147     return *this;
148  }
```

```
149
```

11.6.16 Overloading the Binary Stream Extraction (`>>`) and Stream Insertion (`<<`) Operators

You can input and output fundamental-type data using the **stream extraction operator** (`>>`) and the **stream insertion operator** (`<<`). The C++ standard library overloads these operators for each fundamental type, including pointers and `char*` strings. You also can overload these to perform input and output for custom types.

Line 10 of Fig. 11.4

```
friend std::istream& operator>>(std::istream& in, MyArray& a);
```

and line 58 of Fig. 11.4

```
std::ostream& operator<<(std::ostream& out, const MyArray& a);
```

Perf declare the non-member **overloaded stream-extraction operator (`>>`)** and **overloaded stream-insertion operator** (`<<`). We declared **`operator>>`** in the class as a `friend` because it will access a `MyArray`'s `private` data directly for performance. The **`operator<<`** is not declared as a `friend`. As you'll see, it calls `MyArray`'s `toString` member function to get a `MyArray`'s `string` representation, then outputs it.

Implementing the Stream Extraction Operator

The **overloaded stream-extraction operator (`>>`)** (lines 152–160) takes as arguments an `istream` reference and a `MyArray` reference. It returns the `istream` reference argument to enable **cascaded inputs**, like

```
std::cin >> ints1 >> ints2;
```

When the compiler sees an expression like `cin >> ints1`, it invokes **non-member function `operator>>`** with the call

```
operator>>(std::cin, ints1)
```

We'll say why this needs to be a **non-member function** momentarily. When this call completes, it returns a reference to `cin`, which would then be used in the `cin` statement above to input values into `ints2`. The function creates a `span` (line 153) representing `MyArray`'s dynamically allocated `int` array. Then lines 155–157 iterate through the `span`'s elements, reading one value at a time from the input stream and placing the value in the corresponding element of the dynamically allocated `int` array.

```
150  // overloaded input operator for class MyArray;
151  // inputs values for entire MyArray
152  std::istream& operator>>(std::istream& in, MyArray& a) {
153     std::span<int> items{a.m_ptr.get(), a.m_size};
```

```
154
155     for (auto& item : items) {
156        in >> item;
157     }
158
159     return in; // enables cin >> x >> y;
160  }
161
```

Implementing the Stream Insertion Operator

The *overloaded stream insertion function* (`<<`) (lines 163–166) receives an `ostream` reference and a `const MyArray` reference as arguments and returns an `ostream` reference. The function calls `MyArray`'s `toString` member function, then outputs the resulting `string`. The function returns the `ostream` reference argument to enable **cascaded output statements**, like

```
std::cout << ints1 << ints2;
```

When the compiler sees an expression like `std::cout << ints1`, it invokes **non-member function `operator<<`** with the call

```
operator<<(std::cout, ints1)
```

```
162  // overloaded output operator for class MyArray
163  std::ostream& operator<<(std::ostream& out, const MyArray& a) {
164     out << a.toString();
165     return out; // enables std::cout << x << y;
166  }
167
```

Why `operator>>` and `operator<<` Must Be Non-member Functions

The **`operator>>`** and **`operator<<`** functions are defined as **non-member functions**, so we can specify their operands' order in each function's parameter list. In a binary overloaded operator implemented as a **non-member function**, the first parameter is the left operand, and the second is the right operand.

For the operators **`>>`** and **`<<`**, **the** `MyArray` object should be each operator's *right* operand, so you can use them the way C++ programmers expect, as in the statements

```
std::cin >> ints4;
std::cout << ints4;
```

If we defined these functions as `MyArray` member functions, programmers would have to write the following awkward statements to input or output `MyArray` objects:

```
ints4 >> std::cin;
ints4 << std::cout;
```

Such statements would be confusing and, in some cases, would lead to compilation errors. Programmers are familiar with `cin` and `cout` always appearing to the *left* of these operators.

SE **Overloaded binary operators may be member functions only for the class of the operator's *left* operand. For `operator>>` and `operator<<` to be member functions, we'd have to modify the standard library classes `istream` and `ostream`, which is not allowed.**

Choosing Member vs. Non-Member Functions

Overloaded operator functions, and functions in general, can be

- member functions with direct access to the class's internal implementation details,
- `friend` functions with direct access to the class's internal implementation details or
- non-member, non-`friend` functions—often called **free functions**—that interact with objects of the class through its `public` interface.

CG The C++ Core Guidelines say a function should be a member only if it needs direct access to the class's internal implementation details, such as its `private` data.[35]

35. C++ Core Guidelines, "C.4: Make a Function a Member Only If It Needs Direct Access to the Representation of a Class." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-member.

Another reason to use non-member functions is to define **commutative operators**. Consider a class `HugeInt` for arbitrary-sized integers. With a `HugeInt` named `bigInt`, we might write expressions like

```
bigInt + 7
7 + bigInt
```

Each would sum an `int` value and a `HugeInt`. Like the built-in + operator for fundamental types, each would produce a temporary `HugeInt` containing the sum. To support these expressions, you define two versions of `operator+`, typically as non-member `friend` functions:

```
friend HugeInt operator+(const HugeInt& left, int right);
friend HugeInt operator+(int left, const HugeInt& right);
```

To avoid code duplication, the second function typically would call the first.

11.6.17 `friend` Function `swap`

Line 13 of Fig. 11.4

```
friend void swap(MyArray& a, MyArray& b) noexcept;
```

CG declares the `swap` function used by the **copy assignment operator** to implement the **copy-and-swap idiom**. This function is declared `noexcept`—exchanging the contents of two existing objects does not allocate new resources, so it should not fail.[36] The function (lines 169–172) receives two `MyArray`s. It uses the **standard library `swap` function** to exchange the contents of each object's `m_size` members. It uses the `unique_ptr`'s **`swap` member function** to exchange the contents of each object's `m_ptr` members.

36. C++ Core Guidelines, "C.84: A `swap` Function May Not Fail." Accessed January 15, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-swap-fail.

```
168  // swap function used to implement copy-and-swap copy assignment operator
169  void swap(MyArray& a, MyArray& b) noexcept {
170     std::swap(a.m_size, b.m_size); // swap using std::swap
171     a.m_ptr.swap(b.m_ptr); // swap using unique_ptr swap member function
172  }
```

20 11.7 C++20 Three-Way Comparison Operator (`<=>`)

You'll often compare objects of your custom class types. For example, to sort objects into ascending or descending order using the standard library's `sort` function (Section 6.12; more details in Chapter 14), the objects must be comparable. To support comparisons, you can overload the equality (`==` and `!=`) and relational (`<`, `<=`, `>` and `>=`) operators for your classes. A common practice is to define the `<` and `==` operator functions, then define `!=`, `<=`, `>` and `>=` in terms of `<` and `==`. For instance, for a class `Time` that represents the time of day, its `<=` operator could be implemented as an inline member function in terms of the `<` operator:

```
bool operator<=(const Time& right) const {
   return !(right < *this);
}
```

This leads to lots of "boilerplate" code in which the only difference in the definitions of the overloaded `!=`, `<=`, `>` and `>=` operators among classes is the argument type.

For most types, the compiler can handle the comparison operators for you via the compiler-generated `default` implementation of C++20's **three-way comparison operator (<=>)**,[37,38,39] which is also referred to as the spaceship operator[40] and requires the header **`<compare>`**. Figure 11.6 demonstrates `<=>` for a class `Time` (lines 8–23), which contains

```
1  // fig11_06.cpp
2  // C++20 three-way comparison (spaceship) operator.
3  #include <compare>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <string>
7
8  class Time {
9  public:
10    Time(int hr, int min, int sec) noexcept
11       : m_hr{hr}, m_min{min}, m_sec{sec} {}
12
13    std::string toString() const {
14       return fmt::format("hr={}, min={}, sec={}", m_hr, m_min, m_sec);
15    }
16
17    // <=> operator automatically supports equality/relational operators
18    auto operator<=>(const Time& t) const noexcept = default;
19 private:
20    int m_hr{0};
21    int m_min{0};
22    int m_sec{0};
23 };
24
```

```
25  int main() {
26     const Time t1(12, 15, 30);
27     const Time t2(12, 15, 30);
28     const Time t3(6, 30, 0);
29
30     std::cout << fmt::format("t1: {}\nt2: {}\nt3: {}\n\n",
31                 t1.toString(), t2.toString(), t3.toString());
32
```

```
t1: hr=12, min=15, sec=30
t2: hr=12, min=15, sec=30
t3: hr=6, min=30, sec=0
```

Fig. 11.6 C++20 three-way comparison (spaceship) operator.

37. "C++ Russia 2018: Herb Sutter, New in C++20: The Spaceship Operator," YouTube Video, June 25, 2018. Accessed January 15, 2022. https://www.youtube.com/watch?v=ULkwKsag0Yk.
38. Sy Brand, "Spaceship Operator," August 23, 2018. Accessed January 15, 2022. https://blog.tartanllama.xyz/spaceship-operator/.
39. Cameron DaCamara, "Simplify Your Code with Rocket Science: C++20's Spaceship Operator," June 27, 2019. Accessed January 15, 2022. https://devblogs.microsoft.com/cppblog/simplify-your-code-with-rocket-science-c20s-spaceship-operator/.
40. "Spaceship operator" was coined by Randal L. Schwartz when he was teaching the same operator in a Perl programming course—the operator reminded him of a spaceship in an early video game. https://groups.google.com/a/dartlang.org/g/misc/c/WS5xftItpl4/m/jcIttrMq8agJ?pli=1.

- a constructor (lines 10–11) to initialize its `private` data members (lines 20–22),
- a `toString` function (lines 13–15) to create a `Time`'s `string` representation and
- a **`defaulted` definition of the overloaded `<=>` operator** (line 18).

SE **The `default` compiler-generated `operator<=>` works for any class containing data members that all support the equality and relational operators.** Also, for classes containing built-in arrays as data members, the compiler applies the overloaded `operator<=>` element-by-element as it compares two objects of the arrays' element type.

In `main`, lines 26–28 create three `Time` objects that we'll use in various comparisons, then display their `string` representations. `Time` objects `t1` and `t2` both represent 12:15:30 PM, and `t3` represents 6:30:00 AM. The rest of this program is broken into smaller pieces for discussion purposes.

With `<=>`, the Compiler Supports All the Comparison Operators

When you let the compiler generate the overloaded three-way comparison operator (`<=>`) for you, your class supports all the relational and equality operators, which we demonstrate in lines 34–46. In each case, the compiler rewrites an expression like

```
t1 == t2
```

into an expression that uses the <=> operator, as in

```
(t1 <=> t2) == 0
```

The expression `t1 <=> t2` evaluates to `0` if the objects are equal, a negative value if `t1` is less than `t2` and a positive value if `t1` is greater than `t2`. As you can see, lines 34–46 compiled and produced correct outputs, even though class `Time` did not define any equality or relational operators.

```
33  // using the equality and relational operators
34  std::cout << fmt::format("t1 == t2: {}\n", t1 == t2);
35  std::cout << fmt::format("t1 != t2: {}\n", t1 != t2);
36  std::cout << fmt::format("t1 < t2: {}\n", t1 < t2);
37  std::cout << fmt::format("t1 <= t2: {}\n", t1 <= t2);
38  std::cout << fmt::format("t1 > t2: {}\n", t1 > t2);
39  std::cout << fmt::format("t1 >= t2: {}\n\n", t1 >= t2);
40
41  std::cout << fmt::format("t1 == t3: {}\n", t1 == t3);
42  std::cout << fmt::format("t1 != t3: {}\n", t1 != t3);
43  std::cout << fmt::format("t1 < t3: {}\n", t1 < t3);
44  std::cout << fmt::format("t1 <= t3: {}\n", t1 <= t3);
45  std::cout << fmt::format("t1 > t3: {}\n", t1 > t3);
46  std::cout << fmt::format("t1 >= t3: {}\n\n", t1 >= t3);
47
```

```
t1 == t2: true
t1 != t2: false
t1 < t2: false
t1 <= t2: true
t1 > t2: false
t1 >= t2: true

t1 == t3: false
t1 != t3: true
t1 < t3: false
t1 <= t3: false
t1 > t3: true
t1 >= t3: true
```

Using `<=>` Explicitly

You also can use `<=>` in expressions. An `<=>` expression's result is not convertible to `bool`, so you must compare it to 0 to use `<=>` in a condition, as shown in lines 49, 53 and 57.

```
48     // using <=> to perform comparisons
49     if ((t1 <=> t2) == 0) {
50        std::cout << "t1 is equal to t2\n";
51     }
52
53     if ((t1 <=> t3) > 0) {
54        std::cout << "t1 is greater than t3\n";
55     }
56
57     if ((t3 <=> t1) < 0) {
58        std::cout << "t3 is less than t1\n";
59     }
60  }
```

```
t1 is equal to t2
t1 is greater than t3
t3 is less than t1
```

11.8 Converting Between Types

Most programs process information of many types. Sometimes all the operations "stay within a type." For example, adding an `int` to an `int` produces an `int`. It's often necessary, however, to convert data of one type to data of another type. This can happen in assignments, calculations, passing values to functions and returning values from functions. The compiler knows how to perform certain conversions among fundamental types. You can use cast operators to force conversions among fundamental types.

But what about user-defined types? The compiler does not know how to convert among user-defined types or between user-defined types and fundamental types. You must specify how to do this. Such conversions can be performed with **conversion constructors**. These constructors are called with one argument (we refer to these as **single-argument constructors**). Such constructors can turn objects of other types (including fundamental types) into objects of a particular class.

Conversion Operators

A **conversion operator** (also called a **cast operator**) also can convert an object of a class to another type. Such a conversion operator must be a *non-static member function*. In the `MyArray` case study, we implemented an overloaded conversion operator that converted a `MyArray` to a `bool` value to determine whether the `MyArray` contained elements.

Implicit Calls to Cast Operators and Conversion Constructors

SE **A feature of cast operators and conversion constructors is that the compiler can call them *implicitly* to create objects**. For example, you saw that when an object of our class `MyArray` appears in a program where a `bool` is expected, such as

```
if (ints1) { // if expects a condition
    ...
}
```

the compiler can call the overloaded cast-operator function `operator bool` to convert the object into a `bool` and use the resulting `bool` in the expression.

SE When a conversion constructor or conversion operator is used to perform an implicit conversion, **C++ can apply only one implicit constructor or operator function call** (i.e., a single user-defined conversion) **per expression to try to match the needs of that expression**. The compiler will not satisfy an expression's needs by performing a series of implicit, user-defined conversions.

11.9 `explicit` Constructors and Conversion Operators

SE Recall that we've been declaring as `explicit` every constructor that can be called with one argument, including multiparameter constructors for which we specify default arguments. **Except for copy and move constructors, any constructor that can be called with a *single argument* and is *not* declared `explicit` can be used by the compiler to perform an *implicit conversion*.** The constructor's argument is converted to an object of the class in which the constructor is defined. The conversion is automatic—a cast is not required.

In some situations, implicit conversions are undesirable or error-prone. For example, our `MyArray` class defines a constructor that takes a single `size_t` argument. The intent of this

constructor is to create a `MyArray` object containing a specified number of elements. However, if this constructor were not declared `explicit`, it could be misused by the compiler to perform an **implicit conversion**.

Err **Unfortunately, the compiler might use implicit conversions in cases that you do not expect, resulting in execution-time logic errors or ambiguous expressions that generate compilation errors.**

Accidentally Using a Single-Argument Constructor as a Conversion Constructor

The program (Fig. 11.7) uses Section 11.6's `MyArray` class to demonstrate an improper implicit conversion. To allow this implicit conversion, we removed the `explicit` keyword from line 16 in `MyArray.h` (Fig. 11.4).

```
1  // fig11_07.cpp
2  // Single-argument constructors and implicit conversions.
3  #include <iostream>
4  #include "MyArray.h"
5  using namespace std;
6
7  void outputArray(const MyArray&); // prototype
8
9  int main() {
10    MyArray ints1(7); // 7-element MyArray
11    outputArray(ints1); // output MyArray ints1
12    outputArray(3); // convert 3 to a MyArray and output the contents
13 }
14
15 // print MyArray contents
16 void outputArray(const MyArray& arrayToOutput) {
17    std::cout << "The MyArray received has " << arrayToOutput.size()
18       << " elements. The contents are: " << arrayToOutput << "\n";
19 }
```

```
MyArray(size_t) constructor
The MyArray received has 7 elements. The contents are: {0, 0, 0, 0, 0, 0, 0}
MyArray(size_t) constructor
The MyArray received has 3 elements. The contents are: {0, 0, 0}
MyArray destructor
MyArray destructor
```

Fig. 11.7 Single-argument constructors and implicit conversions.

Line 10 in `main` (Fig. 11.7) instantiates `MyArray` object `ints1` and calls the *single-argument constructor* with the value `7` to specify `ints1`'s number of elements. Recall that the `MyArray` constructor that receives a `size_t` argument initializes all the `MyArray` elements to 0. Line 11 calls function `outputArray` (defined in lines 16–19), which receives as its argument a `const MyArray&`. The function outputs its argument's number of elements and contents. In this case, the `MyArray`'s size is `7`, so `outputArray` displays seven `0`s.

Line 12 calls `outputArray` with the value `3` as an argument. This program does *not* contain an `outputArray` function that takes an `int` argument. So, the compiler determines whether the argument `3` can be converted to a `MyArray` object. Because class `MyArray` provides a constructor with one `size_t` argument (which can receive an `int`) and that constructor is not declared `explicit`, the

compiler assumes the constructor is a **conversion constructor** and uses it to convert the argument `3` into a temporary `MyArray` object containing three elements. Then, the compiler passes this temporary `MyArray` object to function `outputArray`, which displays the temporary `MyArray`'s size and contents. Thus, even though we do not *explicitly* provide an `outputArray` function that receives an `int`, the compiler can compile line 12. The output shows the contents of the three-element `MyArray` containing `0`s.

Preventing Implicit Conversions with Single-Argument Constructors

SE The reason we've declared every single-argument constructor **`explicit` is to *suppress implicit conversions via conversion constructors when such conversions should not be allowed*. A constructor that's declared `explicit` *cannot* be used in an *implicit* conversion.**
Our next program uses class `MyArray` from Section 11.6, which included the keyword `explicit` in the declaration of its **single-argument constructor** that receives a `size_t`:

```
explicit MyArray(size_t size);
```

Figure 11.8 presents a slightly modified version of the program in Fig. 11.7. When this program in Fig. 11.8 is compiled, the compiler produces an error message, such as

```
error: invalid initialization of reference of type ‘const MyArray&’
from expression of type ‘int’
```

```
1  // fig11_08.cpp
2  // Demonstrating an explicit constructor.
3  #include <iostream>
4  #include "MyArray.h"
5  using namespace std;
6
7  void outputArray(const MyArray&); // prototype
8
9  int main() {
10    MyArray ints1{7}; // 7-element MyArray
11    outputArray(ints1); // output MyArray ints1
12    outputArray(3); // convert 3 to a MyArray and output its contents
13    outputArray(MyArray(3)); // explicit single-argument constructor call
14 }
15
16 // print MyArray contents
17 void outputArray(const MyArray& arrayToOutput) {
18    std::cout << "The MyArray received has " << arrayToOutput.size()
19       << " elements. The contents are: " << arrayToOutput << "\n";
20 }
```

Fig. 11.8 Demonstrating an `explicit` constructor.

on g++, indicating that the integer value passed to `outputArray` in line 12 *cannot* be converted to a `const MyArray&`. Line 13 demonstrates how the `explicit` constructor can be used to create a temporary `MyArray` of `3` elements and pass it to `outputArray`.

SE 20 **You should always use the `explicit` keyword on single-argument constructors unless they're intended to be used implicitly as conversion constructors.** In this case, you should declare the single-argument constructor `explicit(false)`. This documents for your class's users that you wish to allow implicit conversions to be performed with that constructor. This new use of `explicit` was introduced in C++20.

11 C++11 `explicit` Conversion Operators

SE 20 **Just as you can declare single-argument constructors `explicit`, you can declare conversion operators `explicit`—or in C++20, `explicit(true)`—to prevent the compiler from using them to perform implicit conversions.** For example, in class `MyArray`, the prototype

```
explicit operator bool() const noexcept;
```

declares the `bool` cast operator `explicit`, so you'd generally have to invoke it explicitly with `static_cast`, as in

```
static_cast<bool>(myArrayObject)
```

As we showed in the `MyArray` case study, C++ can still perform contextual conversions using an `explicit bool` conversion operator to convert an object to a `bool` value in a condition.

11.10 Overloading the Function Call Operator ()

Overloading the **function-call operator `()`** is powerful because functions can take an arbitrary number of comma-separated parameters. We demonstrate the overloaded function-call operator in a natural context in Section 14.5.

11.11 Wrap-Up

This chapter demonstrated how to craft valuable classes, using operator overloading to enable C++'s existing operators to work with custom class objects. First, we used several `string`-class overloaded operators. Next, we presented operator-overloading fundamentals, including which operators can be overloaded and various rules and restrictions on operator overloading.

We introduced dynamic memory management with operators `new` and `delete`, which acquire and release the memory for objects and built-in, pointer-based arrays at runtime. We discussed the problems with using old-style `new` and `delete` statements, such as forgetting to use `delete` to release memory that's no longer needed. We introduced RAII (Resource Acquisition Is Initialization) and demonstrated smart pointers. You saw how to dynamically allocate memory as you created a `unique_ptr` smart pointer object. The `unique_ptr` automatically released the memory when the object went out of scope, preventing a memory leak.

Next, we presented the chapter's substantial capstone `MyArray` case study, which used overloaded operators and other capabilities to solve various problems with pointer-based arrays. We implemented the five special member functions typically defined in classes that manage their own resources—the copy constructor, copy assignment operator, move constructor, move assignment operator and destructor. We also discussed how to autogenerate these special member functions

with `= default` and remove them with `= delete`. The class overloaded many operators and defined a conversion from `MyArray` to `bool` for determining whether a `MyArray` is empty or contains elements.

We introduced C++20's new three-way comparison operator (<=>)—also called the spaceship operator. You saw that for some classes, the default compiler-generated `<=>` operator enables a class to support all six equality and relational operators without you having to explicitly overload them. We discussed in more detail converting between types, problems with implicit conversions defined by single-argument constructors and how to prevent those problems with the keyword `explicit`. Finally, we discussed overloading the function-call operator, `()`, which we'll demonstrate in later chapters.

We've introduced some exception-handling fundamentals. The next chapter discusses exception handling in detail, so you can create more robust and fault-tolerant applications that can deal with problems and continue executing, or terminate gracefully. We'll show how to handle exceptions if operator `new` fails to allocate memory for an object. We'll also introduce several C++ standard library exception-handling classes and show how to create your own.

12. Exceptions and a Look Forward to Contracts

Objectives

In this chapter, you'll:

- Understand the exception-handling flow of control with `try`, `catch` and `throw`.
- Provide exception guarantees for your code.
- Understand the standard library exception hierarchy.
- Define a custom exception class.
- Understand how stack unwinding enables exceptions not caught in one scope to be caught in an enclosing scope.
- Handle dynamic memory allocation failures.
- Catch exceptions of any type with `catch(...)`.
- Understand what happens with uncaught exceptions.
- Understand which exceptions should not be handled and which cannot be handled.
- Understand why some organizations disallow exceptions and the impact that can have on software development efforts.
- Understand the performance costs of exception handling.
- Look ahead to how contracts can eliminate many use-cases of exceptions, enabling more functions to be `noexcept`.

Outline

12.1 Introduction
12.2 Exception-Handling Flow of Control
12.2.1 Defining an Exception Class to Represent the Type of Problem That Might Occur
12.2.2 Demonstrating Exception Handling
12.2.3 Enclosing Code in a `try` Block
12.2.4 Defining a `catch` Handler for `DivideByZeroExceptionS`
12.2.5 Termination Model of Exception Handling
12.2.6 Flow of Control When the User Enters a Nonzero Denominator
12.2.7 Flow of Control When the User Enters a Zero Denominator
12.3 Exception Safety Guarantees and `noexcept`
12.4 Rethrowing an Exception
12.5 Stack Unwinding and Uncaught Exceptions
12.6 When to Use Exception Handling
12.6.1 assert Macro
12.6.2 Failing Fast
12.7 Constructors, Destructors and Exception Handling
12.7.1 Throwing Exceptions from Constructors
12.7.2 Catching Exceptions in Constructors via Function `try` Blocks
12.7.3 Exceptions and Destructors: Revisiting `noexcept(false)`
12.8 Processing `new` Failures
12.8.1 `new` Throwing `bad_alloc` on Failure
12.8.2 `new` Returning `nullptr` on Failure
12.8.3 Handling `new` Failures Using Function `set_new_handler`
12.9 Standard Library Exception Hierarchy

12.10 C++'s Alternative to the `finally` Block: Resource Acquisition Is Initialization (RAII)
12.11 Some Libraries Support Both Exceptions and Error Codes
12.12 Logging
12.13 Looking Ahead to Contracts
12.14 Wrap-Up

12.1 Introduction

C++ is used to build real-world, mission-critical and business-critical software. Bjarne Stroustrup (C++'s creator) maintains on his website an extensive list[1] of about 150 applications and systems written partially or entirely in C++. Here are just a few:

1. Bjarne Stroustrup, "C++ Applications," October 27, 2020. Accessed January 16, 2022. https://www.stroustrup.com/applications.html.

- portions of most major operating systems, like Apple macOS and Microsoft Windows,
- all the compilers we use in this book (GNU g++, Clang and Visual C++),
- Amazon.com,
- aspects of Facebook that require high performance and reliability,
- Bloomberg's real-time financial information systems,
- many of Adobe's authoring, graphics and multimedia applications,
- various database systems, such as MongoDB and MySQL,
- many NASA projects, including aspects of the software in the Mars rovers,
- and much more.

For another extensive list of over 100 applications and systems, see *The Programming Languages Beacon*.[2]

2. Vincent Lextrait, "The Programming Languages Beacon v16," March 2016. Accessed January 16, 2022. https://www.mentofacturing.com/vincent/implementations.html.

Many of these systems are massive. Consider some statistics from the "Codebases: Millions of Lines of Code Infographic" (which is not C++ specific):[3]

3. "Codebases: Millions of Lines of Code Infographic," September 24, 2015. Accessed January 16, 2022. https://www.informationisbeautiful.net/visualizations/million-lines-of-code/. For a spreadsheet of the infographic's data sources, see http://bit.ly/CodeBasesInfographicData.

- A Boeing 787 aircraft's avionics and online support systems have 6.5 million lines of code, and its flight software is 14 million lines of code.
- The F-35 Fighter Jet has 24 million lines of code.
- The Large Hadron Collider—the world's largest particle accelerator[4]—in Geneva, Switzerland, has 50 million lines of code.

4. “The Large Hadron Collider.” Accessed January 16, 2022. https://home.cern/science/accelerators/large-hadron-collider.

- Facebook has 62 million lines of code.
- An average modern high-end car has 100 million lines of code, and they’re expected to have 300 million lines of code by 2030[5]—across hundreds of processors and controllers.[6]

5. Anthony Martin, “Vehicle Cybersecurity: Control the Code, Control the Road,” March 18, 2020. Accessed January 16, 2022. https://www.vehicledynamicsinternational.com/features/vehicle-cybersecurity-control-the-code-control-the-road.html.

6. Ferenc Valenta’s answer to “How many lines of code are in a car?” January 10, 2019. Accessed January 16, 2022. https://www.quora.com/How-many-lines-of-code-are-in-a-car/answer/Ferenc-Valenta.

Self-driving cars are expected to require one billion lines of code.[7] For large and small codebases alike, it’s essential to eliminate bugs during development and handle problems that may occur once the software is deployed in real products. That is the focus of this chapter.

7. “Jaguar Land Rover Finds the Teenagers Writing the Code for a Self-Driving Future,” April 15, 2019. Accessed January 16, 2022. https://media.jaguarlandrover.com/news/2019/04/jaguar-land-rover-finds-teenagers-writing-code-self-driving-future.

Exceptions and Exception Handling

SE As you know, an **exception** indicates a problem that occurs during a program’s execution. Exceptions may surface through

- explicitly mentioned code in a `try` block,
- calls to other functions (including library calls) and
- operator errors, like `new` failing to acquire additional memory at execution time (Section 12.8).

Exception handling helps you write robust, **fault-tolerant programs** that catch infrequent problems and

- deal with them and continue executing,
- perform appropriate cleanup for exceptions that cannot or should not be handled and terminate gracefully or
- terminate abruptly in the case of unanticipated exceptions—a concept called failing fast, which we discuss in Section 12.6.2.

Exception Handling, Stack Unwinding and Rethrowing Exceptions

You've seen exceptions get thrown (Section 6.15) and `try`...`catch` used to handle them (Section 9.7.11). This chapter reviews these exception-handling concepts in an example that demonstrates the flows of control

- when a program executes successfully and
- when an exception occurs.

We discuss use-cases for catching then rethrowing exceptions, and show how C++ handles an exception that is not caught in a particular scope. We introduce logging exceptions into a file or database that developers can review later for debugging purposes.

When to Use Exceptions and Exception Safety Guarantees

Exceptions are not for all types of error handling, so we discuss when and when not to use them. We also introduce the exception safety guarantees you can provide in your code—from none at all to indicating with `noexcept` that your code does not throw exceptions.

Exceptions in the Context of Constructors and Destructors

We discuss why exceptions are used to indicate errors during construction and why destructors should not throw exceptions. We also demonstrate how to use function `try` blocks to catch exceptions from a constructor's member-initializer list.

Handling Dynamic Memory Allocation Failures

By default, `new` throws exceptions when dynamic memory allocation fails. We demonstrate how to catch such `bad_alloc` exceptions. We also show how dynamic memory allocation failures were handled in legacy code before `bad_alloc` was added to C++.

Standard Library Exception Hierarchy and Custom Exception Classes

We introduce the C++ standard library exception-handling class hierarchy. We create a custom exception class that inherits from one of the C++ standard library exception classes. You'll see why it's important to catch exceptions by reference to enable exception handlers to catch derived-class exception types.

Exceptions Are Not Universally Used

Most C++ features have a zero-overhead principle in which you do not pay a price for a given feature unless you use it.[8] Exceptions violate this principle—programs with exception handling have a larger memory footprint. Some organizations disallow exception handling for this and other reasons. We'll discuss why some libraries provide dual interfaces, enabling developers to choose whether to use versions of functions that throw exceptions or versions that set error codes.

---

8. Herb Sutter, "De-fragmenting C++: Making Exceptions and RTTI More Affordable and Usable," September 23, 2021. Accessed January 16, 2022. https://www.youtube.com/watch?v=ARYP83yNAWk.

Looking Ahead to Contracts

The chapter concludes with an introduction to contracts. This feature was originally targeted for C++20 but delayed to a future version. We'll introduce preconditions, postconditions and assertions, which you'll see are implemented as contracts tested at runtime. If such conditions fail, a contract

violation occurs. By default, the code terminates immediately, enabling you to find errors faster, eliminate them during development and, hopefully, create more robust code for deployment. You'll test the example code using GCC's experimental contracts implementation on https://godbolt.org.

Exceptions Enable You to Separate Error Handling from Program Logic

SE SE Exception handling provides a standard mechanism for processing errors. This is especially important when working on a large project. As you'll see, using `try`...`catch` lets you separate the successful path of execution from the error path of execution,[9,10] making your code easier to read and maintain.[11] Also, once an exception occurs, it cannot be ignored—in Section 12.5, you'll see that ignoring an exception can lead to program termination.[12,13]

---

9. "Technical Report on C++ Performance," February 15, 2006. Accessed January 16, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (page 34).

10. "What Does It Mean That Exceptions Separate the Good Path (or Happy Path) from the Bad Path?" Accessed January 16, 2022. https://isocpp.org/wiki/faq/exceptions#exceptions-separate-good-and-bad-path.

11. Barbara Thompson, "C++ Exception Handling: Try, Catch, Throw Example," updated January 1, 2022. Accessed January 16, 2022. https://www.guru99.com/cpp-exceptions-handling.html.

12. "Technical Report on C++ Performance," February 15, 2006. Accessed January 16, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (page 34).

13. Manoj Piyumal, "Some Useful Facts to Know Before Using C++ Exceptions," December 5, 2017. Accessed January 16, 2022. https://dzone.com/articles/some-useful-facts-to-know-when-using-c-exceptions.

Indicating Errors via Return Values

Err Without exception handling, it's common for a function to calculate and return a value on success or return an error indicator on failure:

- A problem with this architecture is using the return value in a subsequent calculation without first checking whether the value is the error indicator.
- Also, there are cases in which returning error codes is not possible, such as in constructors and overloaded operators.

Exception handling eliminates these problems.

12.2 Exception-Handling Flow of Control

Let's demonstrate the flow of control:

- when a program executes successfully and
- when an exception occurs.

For demo purposes, Figs. 12.1–12.2 show how to deal with a common arithmetic problem—division-by-zero. In C++, division-by-zero in both integer and floating-point arithmetic is undefined behavior. Each of our preferred compilers issues warnings or errors if they detect integer division-by-zero at compile-time. Visual C++ also issues an error if it detects floating-point division-by-zero. At runtime, integer division-by-zero usually causes a program to crash. Some C++ implementations allow floating-point division-by-zero and produce positive or negative infinity—displayed as `inf` or `-inf`, respectively. This is true for each of our preferred compilers.

The example consists of two files:

- `DivideByZeroException.h` (Fig. 12.1) defines a **custom exception class** representing the type of problem that might occur in the example.
- `fig12_02.cpp` (Fig. 12.2) defines the `quotient` function and the `main` function that calls it. We'll use these to explain the exception-handling flow of control.

```
1  // Fig. 12.1: DivideByZeroException.h
2  // Class DivideByZeroException definition.
3  #include <stdexcept> // stdexcept header contains runtime_error
4
5  // DivideByZeroException objects should be thrown
6  // by functions upon detecting division-by-zero
7  class DivideByZeroException : public std::runtime_error {
8  public:
9     // constructor specifies default error message
10    DivideByZeroException()
11       : std::runtime_error{"attempted to divide by zero"} {}
12 };
```

Fig. 12.1 Class `DivideByZeroException` definition.

12.2.1 Defining an Exception Class to Represent the Type of Problem That Might Occur

Figure 12.1 defines `DivideByZeroException`—a custom exception class that our program will throw when it detects attempts to divide by zero. We defined this as a derived class of standard library class `runtime_error` (from header `<stdexcept>`). A typical derived class of `runtime_error` defines only a constructor (e.g., lines 10–11) that passes an error-message string to the base-class constructor.

SE CG Generally, you should derive your custom exception classes from those in the C++ standard library and name your class, so it's clear what problem occurred. The C++ Core Guidelines indicate that such exception classes are less likely to be confused with exceptions thrown by other libraries, like the C++ standard library.[14] We'll say more about the standard exception classes in Section 12.9.

14. C++ Core Guidelines, "E.14: Use Purpose-Designed User-Defined Types as Exceptions (not Builtin Types)." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-types.

12.2.2 Demonstrating Exception Handling

Figure 12.2 uses exception handling to wrap code that might throw a `DivideByZeroException` and to handle that exception should one occur. Function `quotient` receives two `double`s, divides the first by the

second and returns the `double` result. Function `quotient` treats all attempts to divide by zero as errors. If it determines the second argument is zero, it **throws a `DivideByZeroException`** (line 13) to indicate this problem to the caller. **The operand of a `throw` can be of any copy-constructible type**, not just objects of types that directly or indirectly derive from `exception`. Copy-constructible types are required because the exception mechanism copies the exception into a temporary exception object.[15]

15. "throw Expression." Accessed January 16, 2022. https://en.cppreference.com/w/cpp/language/throw.

```
1  // fig12_02.cpp
2  // Example that throws an exception on
3  // an attempt to divide by zero.
4  #include <fmt/format.h>
5  #include <iostream>
6  #include "DivideByZeroException.h" // DivideByZeroException class
7
8  // performs division only if the denominator is not zero;
9  // otherwise, throws DivideByZeroException object
10 double quotient(double numerator, double denominator) {
11     // throw DivideByZeroException if trying to divide by zero
12     if (denominator == 0.0) {
13        throw DivideByZeroException{};
14     }
15
16     // return division result
17     return numerator / denominator;
18 }
19
20 int main() {
21    int number1{0}; // user-specified numerator
22    int number2{0}; // user-specified denominator
23
24    std::cout << "Enter two integers (end-of-file to end): ";
25
26    // enable user to enter two integers to divide
27    while (std::cin >> number1 >> number2) {
28       // try block contains code that might throw exception
29       // and code that will not execute if an exception occurs
30       try {
31          double result{quotient(number1, number2)};
32          std::cout << fmt::format("The quotient is: {}\n", result);
33       }
34       catch (const DivideByZeroException& divideByZeroException) {
35          std::cout << fmt::format("Exception occurred: {}\n",
36             divideByZeroException.what());
37       }
38
39       std::cout << "\nEnter two integers (end-of-file to end): ";
40    }
41
42    std::cout << '\n';
43 }
Enter two integers (end-of-file to end): 100 7
```

```
The quotient is: 14.2857

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

Enter two integers (end-of-file to end): ^Z
```

Fig. 12.2 Example that throws an exception on an attempt to divide by zero.

Assuming that the user does not specify `0` as the denominator for the division, function `quotient` returns the division result. If the user inputs `0` for the denominator, `quotient` throws an exception. In this case, `main` handles the exception, then asks the user to enter two new values before calling `quotient` again. In this way, the program can continue executing even after an improper value is entered, **making the program more robust**. In the sample output, the first two lines show a successful calculation, and the next two show a failure due to an attempt to divide by zero. After we discuss the code, we'll consider the user inputs and flows of control that yield the outputs shown in Fig. 12.3.

12.2.3 Enclosing Code in a try Block

CG The program prompts the user to enter two integers, then inputs them in the `while` loop's condition (line 27). Line 31 passes the values to `quotient` (lines 10–18), which either divides the integers and returns a result or **throws an exception** if the user attempts to divide by zero. Line 13 in `quotient` is known as the **throw point**. Exception handling is geared to situations where the function that detects an error cannot perform its task.[16]

16. C++ Core Guidelines, "E.2: Throw an Exception to Signal That a Function Can't Perform Its Assigned Task." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-throw.

The `try` block in lines 30–33 encloses two statements:

- the `quotient` function call (line 31), which can throw an exception, and
- the statement that displays the division result (line 32).

Line 32 executes *only* if `quotient` successfully returns a result.

12.2.4 Defining a `catch` Handler for `DivideByZeroException`s

SE At least one `catch` handler (lines 34–37) *must* immediately follow each `try` block. **An exception parameter should be declared as a `const` reference to the exception type to process**—in this case, a `DivideByZeroException`. **This has two key benefits:**

- Perf **It prevents copying the exception as part of the `catch` operation.**
- **It enables catching derived-class exceptions without slicing** (see below).

`catching` By Reference Avoids Slicing

Err When an exception occurs in a `try` block, **C++ executes the first `catch` handler with a parameter of the same type as, or a base-class type of, the thrown exception's type**. If a base-class `catch` handler catches a derived-class exception object *by value*, only the base-class portion of the derived-class exception object will be copied into the exception parameter. This logic error,

known as slicing, occurs when a derived-class object is copied into or assigned to a base-class object. **Always catch exceptions by reference to prevent slicing**.

`catch` Parameter Name

An exception parameter that includes the *optional* parameter name (as in line 34) enables the `catch` handler to interact with the caught exception. Line 36 calls its `what` member function to get the exception's error message.

Tasks Performed in `catch` Handlers

Typical tasks performed by `catch` handlers include

- reporting the error to the user,
- logging errors to a file that developers can study for debugging purposes,
- terminating the program gracefully,
- trying an alternative strategy to accomplish the failed task,
- rethrowing the exception to the current function's caller or
- throwing a different exception type.

This example's `catch` handler simply reports that the user attempted to divide by zero.

Common Errors When Defining `catch` Handlers

Programmers new to exception handling should be aware of several common coding errors:

- Err It's a syntax error to place code between a `try` block and its corresponding `catch` handlers or between its `catch` handlers.
- Err Each `catch` handler can have only one parameter—specifying a comma-separated list of exception parameters is a syntax error.
- Err It's a compilation error to catch the same type in multiple `catch` handlers following a single `try` block.
- Err It's a logic error to catch a base-class exception before a derived-class exception—compilers typically warn you when this occurs.

12.2.5 Termination Model of Exception Handling

If no exceptions occur in a `try` block, program control continues with the first statement after the last `catch` following that `try` block. If an exception does occur, the `try` block terminates immediately—any local variables defined in that block go out of scope. **This is a strength of exception handling. Destructors for the `try` block's local variables are guaranteed to run, enabling programs to avoid resource leaks.**[17] Next, the program searches for and executes the first matching `catch` handler. If execution reaches that `catch` handler's closing right brace (`}`), the exception is considered handled. Any local variables in the `catch` handler, including the `catch` parameter, go out of scope.

17. "Technical Report on C++ Performance," February 15, 2006. Accessed January 16, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf (page 34).

C++ uses the termination model of exception handling, in which **program control cannot return to the throw point**. Instead, control resumes with the first statement (line 39) after the `try` block's last `catch` handler.

If an exception occurs in a function and is not caught there, the function terminates immediately. The program attempts to locate an enclosing `try` block in the calling function. This process, called **stack unwinding**, is discussed in Section 12.5.

12.2.6 Flow of Control When the User Enters a Nonzero Denominator

Consider the flow of control in Fig. 12.3 when the user inputs the numerator `100` and the denominator `7`. In line 12, `quotient` determines that the `denominator` is not `0`, so line 17 performs the division and returns the result (`14.2857`) to line 31. Program control continues sequentially from line 31, so line 32 displays the division result, and control reaches the `try` block's ending brace. In this case, **the `try` block completed successfully, so program control skips the `catch` handler** (lines 34–37) and continues with line 39.

12.2.7 Flow of Control When the User Enters a Zero Denominator

Now consider the flow of control in which the user inputs the numerator `100` and the denominator `0`. In line 12, `quotient` determines that the `denominator` is `0`, so line 13 uses keyword `throw` to create and throw a `DivideByZeroException` object. This calls the `DivideByZeroException` constructor to initialize the exception object. Our exception class's constructor does not have parameters. For exception constructors that do, you'd pass the argument(s) to the constructor as you create the object, as in

```
throw out_of_range{"Index out of range"};
```

SE When the exception is thrown, `quotient` exits immediately without performing the division. If you explicitly throw an exception from your own code, you generally do so *before* the error has an opportunity to occur. That's not always possible. For example, your function might call another function, which encounters an error and throws an exception.

SE We enclosed the `quotient` call (line 31) in a `try` block, so program control enters the first matching `catch` handler (lines 34–37) that immediately follows the `try` block. In this program, that handler catches `DivideByZeroException`s—the type thrown by `quotient`—then prints the error message returned by function `what`. Associating each type of runtime error with an appropriately named exception type improves program clarity.

12.3 Exception Safety Guarantees and noexcept

Client-code programmers need to know what to expect when using your code. Is there a potential for exceptions? If so, what will the program's state be if an exception occurs? When you design your code, consider what **exception safety guarantees** you'll make:[18,19]

18. Klaus Iglberger, "Back to Basics: Exceptions," October 5, 2020. Accessed January 16, 2022 https://www.youtube.com/watch?v=0ojB8c0xUd8.

19. "Exceptions." Accessed January 16, 2022. https://en.cppreference.com/w/cpp/language/exceptions.

- **No guarantee**—If an exception occurs, the program may be in an invalid state and resources, like dynamically allocated memory, could be leaked.

- Basic exception guarantee—**If an exception occurs, objects' states remain valid but possibly modified, and no resources are leaked.** Generally, code that can cause exceptions should provide at least a basic exception guarantee.
- Strong exception guarantee—**If an exception occurs, objects' states remain unmodified or, if objects were modified, they're returned to their original states before the operation that caused the exception.** We implemented the copy assignment operator in Chapter 11's `MyArray` class with a strong exception guarantee via the copy-and-swap idiom. The operator first copies its argument, which could fail to allocate resources. In that case, the assignment fails without modifying the original object; otherwise, the assignment completes successfully.
- No throw exception guarantee—**The operation does not throw exceptions.** For example, in Chapter 11, the `MyArray` class's move constructor, move assignment operator and `friend` function `swap` were declared `noexcept`. They work with existing resources rather than allocating new ones, so they cannot fail. Only functions that truly cannot fail should be declared `noexcept`. If an exception occurs in a `noexcept` function, the program terminates immediately.

12.4 Rethrowing an Exception

Sometimes, you might
- catch an exception,
- partially process it, then
- notify the caller by **rethrowing the exception**.

Doing so indicates that **the exception is not yet handled**. The statement

```
throw;
```

Err executed in a `catch` handler returns the current exception to the next enclosing `try` block. Then a `catch` handler listed after that `try` block attempts to handle the exception. Executing the preceding statement outside a `catch` handler terminates the program immediately.

If the `catch` handler has a parameter name for the exception it catches, such as `ex`, do not rethrow the exception with

```
throw ex;
```

Perf Err This unnecessarily copies the original exception object. This also can be a logic error. If the exception handler's type is a base class of the rethrown exception, **slicing** occurs.

Use-Cases for Rethrowing an Exception

There are various use-cases for rethrowing exceptions:
- You might want to log to a file where each exception occurs for future debugging. In this case, you'd catch the exception, log it, then rethrow the exception for further processing in an enclosing scope. We discuss logging in Section 12.12.
- 11 You might want to throw a different exception type that's more specific to your library or application. In this scenario, you might wrap the original exception into the new exception by using the C++11 `nested_exception` class.[20]

20. "std::nested_exception." Accessed January 16, 2022. https://en.cppreference.com/w/cpp/error/nested_exception.

- You do partial processing of an exception, such as releasing a resource, then rethrow the exception for further processing in a `catch` of an enclosing `try` block.
- In some cases, exceptions are implicitly rethrown, such as in function `try` blocks for constructors and destructors, which we discuss in Section 12.7.2.

Demonstrating Rethrowing an Exception

Figure 12.3 demonstrates *rethrowing* an exception. In `main`'s `try` block (lines 24–28), line 26 calls `throwException` (lines 7–20), which contains a `try` block (lines 9–12) from which the `throw` statement in line 11 throws an `exception` object. The function's `catch` handler (lines 13–17) catches this exception, prints an error message (lines 14–15) and rethrows the exception (line 16). This immediately returns control to line 26 in `main`'s `try` block. That `try` block *terminates* (so line 27 does *not* execute), and the `catch` handler in `main` (lines 29–31) catches the exception and prints an error message (line 30). We do not use the exception parameters in this example's `catch` handlers, so we omit the exception parameter names and specify only the type of exception to catch (lines 13 and 29).

```
1  // fig12_03.cpp
2  // Rethrowing an exception.
3  #include <iostream>
4  #include <exception>
5
6  // throw, catch and rethrow exception
7  void throwException() {
8     // throw exception and catch it immediately
9     try {
10       std::cout << " Function throwException throws an exception\n";
11       throw std::exception{}; // generate exception
12    }
13    catch (const std::exception&) { // handle exception
14       std::cout << " Exception handled in function throwException"
15          << "\n Function throwException rethrows exception";
16       throw; // rethrow exception for further processing
17    }
18
19    std::cout << "This should not print\n";
20 }
21
22 int main() {
23    // call throwException
24    try {
25       std::cout << "main invokes function throwException\n";
26       throwException();
27       std::cout << "This should not print\n";
28    }
29    catch (const std::exception&) { // handle exception
30       std::cout << "\n\nException handled in main\n";
31    }
32
33    std::cout << "Program control continues after catch in main\n";
34 }
```

```
main invokes function throwException
  Function throwException throws an exception
```

```
  Exception handled in function throwException
  Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
```

Fig. 12.3 Rethrowing an exception.

12.5 Stack Unwinding and Uncaught Exceptions

When you do not catch an exception in a particular scope, the function-call stack "unwinds" in an attempt to catch the exception in an **enclosing scope**—that is, a `catch` block of the next outer `try` block. You can nest `try` blocks, in which case the next outer `try` block could be in the same function. When `try` blocks are not nested, stack unwinding occurs. The function in which the exception was not caught terminates, and its existing local variables go out of scope. During stack unwinding, control returns to the statement that invoked the function. If that statement is in a `try` block, that block terminates, and an attempt is made to `catch` the exception. If the statement is not in a `try` block or the exception is not caught, stack unwinding continues. **In fact, this mechanism is one reason you should not wrap a `try…catch` around every function call that might throw an exception.[21] Sometimes it's more appropriate to let an earlier function in the call chain deal with the problem.** Figure 12.4 demonstrates stack unwinding.

21. C++ Core Guidelines, "E.17: Don't Try to Catch Every Exception in Every Function." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-not-always.

```
1  // fig12_04.cpp
2  // Demonstrating stack unwinding.
3  #include <iostream>
4  #include <stdexcept>
5
6  // function3 throws runtime error
7  void function3() {
8     std::cout << "In function 3\n";
9
10    // no try block, stack unwinding occurs, return control to function2
11    throw std::runtime_error{"runtime_error in function3"};
12 }
13
14 // function2 invokes function3
15 void function2() {
16    std::cout << "function3 is called inside function2\n";
17    function3(); // stack unwinding occurs, return control to function1
18 }
19
20 // function1 invokes function2
21 void function1() {
22    std::cout << "function2 is called inside function1\n";
23    function2(); // stack unwinding occurs, return control to main
24 }
25
26 // demonstrate stack unwinding
27 int main() {
```

```
28     // invoke function1
29     try {
30        std::cout << "function1 is called inside main\n";
31        function1(); // call function1 which throws runtime_error
32     }
33     catch (const std::runtime_error& error) { // handle runtime error
34        std::cout << "Exception occurred: " << error.what()
35            << "\nException handled in main\n";
36     }
37  }
```

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

Fig. 12.4 Demonstrating stack unwinding.

In `main`, line 31 in the `try` block calls `function1` (lines 21–24), then `function1` calls `function2` (lines 15–18), which in turn calls `function3` (lines 7–12). Line 11 of `function3` throws a `runtime_error` object—this is the throw point. At this point, control proceeds as follows:

- No `try` block encloses line 11, so stack unwinding begins. `function3` terminates, returning control to line 17 in `function2`.
- No `try` block encloses line 17, so stack unwinding continues. `function2` terminates, returning control to line 23 in `function1`.
- Again, no `try` block encloses line 23, so stack unwinding occurs again. `function1` terminates and returns control to line 31 in `main`.
- The `try` block in lines 29–32 encloses line 31, so the `try` block's first matching `catch` handler (line 33–36) catches and processes the exception by displaying the exception message.

Uncaught Exceptions

Err **If an exception is not caught during stack unwinding, C++ calls the standard library function `terminate`, which calls `abort` to terminate the program.** To demonstrate this, we removed `main`'s `try`...`catch` in `fig12_04.cpp`, keeping only lines 30–31 from the `try` block in Fig. 12.4. The modified version, `fig12_04modified.cpp`, is in the `fig12_04` example folder. When you execute this modified version and the exception is not caught in `main`, the program terminates. The following shows the output when we executed the program using GNU C++—note that the output mentions `terminate` was called:

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
terminate called after throwing an instance of 'std::runtime_error'
  what():  runtime_error in function3
Aborted (core dumped)
```

12.6 When to Use Exception Handling

Exception handling is designed to process infrequent **synchronous errors**, which occur when a statement executes even if your code is correct,[22] such as

22. “Modern C++ Best Practices for Exceptions and Error Handling.” Accessed January 16, 2022. https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp.

- accessing a web service that’s temporarily unavailable,
- attempting to read from a file that does not exist,
- attempting to access a file for which you do not have appropriate permissions,
- dynamic memory allocation failures and more.

Exception handling is not designed to process errors associated with **asynchronous events**, which occur in parallel with and independent of the program’s flow of control. Examples include **I/O completions**, **network message arrivals**, **mouse clicks** and **keystrokes**.

Complex applications usually consist of **predefined software components** (such as standard library classes) and **application-specific components** that use the predefined ones. When a predefined component encounters a problem, it needs to communicate the problem to the application-specific component—**the predefined component cannot know how each application will process a problem**. Sometimes, that problem must be communicated to a function several calls earlier in the function-call chain that led to the exception.

Exception handling provides a single, uniform technique for processing problems. This helps programmers on large projects understand each other’s error-processing code. It also enables predefined software components (such as standard library classes) to communicate problems to application-specific components. You should incorporate your exception-handling strategy into your system from its inception[23]—doing so after a system has been implemented can be difficult.

23. C++ Core Guidelines, “E.1: Develop an Error-Handling Strategy Early in a Design.” Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-design.

When Not to Use Exception Handling

The `isocpp.org` FAQ section on exceptions lists several scenarios in which you should *avoid* using exceptions.[24] These include

24. “What Shouldn’t I Use Exceptions For?” Accessed January 16, 2022. https://isocpp.org/wiki/faq/exceptions#why-not-exceptions.

- **cases in which failures are expected**, such as converting incorrectly formatted strings to numeric values where the strings might not have the correct format;
-
applications that have strict performance requirements where the overhead of throwing exceptions and stack unwinding is unacceptable, such as the real-time systems used in the United States Joint Strike Fighter plane—for which there is a C++ coding standard;[25] and

25. “Joint Strike Fighter Air Vehicle C++ Coding Standards,” December 2005. Accessed January 16, 2022. https://www.stroustrup.com/JSF-AV-rules.pdf.

- **frequent errors that should not happen in code**, such as **accessing out-of-range array elements**, **dereferencing a null pointer** and **division by zero.**[26] Interestingly, the C++ standard includes the `out_of_range` exception class, and various C++ standard library classes, such as `array` and `vector`, have member functions that throw `out_of_range` exceptions.

26. We used division by zero as a mechanical demonstration of the exception handling flow of control.

Perf **Exception handling can increase the executable size of your program,**[27] **which may be unacceptable in memory-constrained devices, such as embedded systems.** According to the `isocpp.org` FAQ on exception handling, however, “exception handling is extremely cheap when you don’t throw an exception. It costs nothing on some implementations. All the cost is incurred when you throw an exception—that is, normal code is faster than code using error-return codes and tests. You incur cost only when you have an error.”[28] For a detailed discussion of exception-handling performance, see Section 5.4 of the **Technical Report on C++ Performance.**[29]

27. Vishal Chovatiya, “C++ Exception Handling Best Practices: 7 Things To Know,” November 3, 2019. Accessed January 16, 2022. http://www.vishalchovatiya.com/7-best-practices-for-exception-handling-in-cpp-with-example/.

28. “Why Use Exceptions?” Accessed January 16, 2022. https://isocpp.org/wiki/faq/exceptions#why-exceptions.

29. “Technical Report on C++ Performance,” February 15, 2006. Accessed January 16, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf.

SE Functions with common errors generally should return `nullptr`, `0` or other appropriate values, such as `bool`s, rather than throw exceptions. A program calling such a function can check the return value to determine whether the function call succeeded or failed. Herb Sutter—the ISO C++ language standards committee (WG21) Convener and a Software Architect at Microsoft—indicates, “Programs bugs are not recoverable run-time errors and so should not be reported as exceptions or error codes.” He goes on to say that the process is already under way to migrate the C++ standard libraries away from throwing exceptions for such errors.[30] In Section 12.13, Looking Ahead to Contracts, we’ll see that the new contracts capabilities, originally scheduled to be included in C++20 and now deferred until at least C++23, can help reduce the need for exceptions in the standard library.

30. Herb Sutter, “Zero-Overhead Deterministic Exceptions: Throwing Values,” August 4, 2019. Accessed January 16, 2022. https://wg21.link/p0709R4.

CG The C++ Core Guidelines indicate that many `try`…`catch` statements in your code can be a sign of too much low-level resource management.[31] To avoid the need for `try`…`catch` in such code, they recommend designing your classes to use **RAII** (Resource Acquisition Is Initialization; Chapter 11). Recall that with RAII, an object’s constructor acquires resources, and its destructor releases them. So, if an object goes out of scope for any reason, including an exception, the object’s resources will be released.

31. C++ Core Guidelines, “E.18: Minimize the Use of Explicit try/catch.” Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-catch.

12.6.1 `assert` Macro

When implementing and debugging programs, it's sometimes useful to state conditions that should be true at a particular point in a function. These conditions, called **assertions**, help ensure a program's validity by catching potential bugs and identifying possible logic errors during development. An assertion is a runtime check for a condition that should always be true if your code is correct. If the condition is false, the program terminates immediately, displaying an error message that includes the filename and line number where the problem occurred and the condition that failed. You implement an assertion using the `assert` **macro** from the `<cassert>` **header**,[32] as in

```
assert (condition);
```

32. "assert." Accessed January 16, 2022. https://en.cppreference.com/w/cpp/error/assert.

Assertions are primarily a development-time aid[33] for alerting programmers to coding errors that need to be fixed. For example, you might use an assertion in a function that processes arrays to ensure that the array indexes are greater than 0 and less than the array's length. Once you're done debugging, you can disable assertions by adding the following preprocessor directive before the `#include` for the `<cassert>` header:

```
#define NDEBUG
```

33. "Modern C++ Best Practices For Exceptions and Error Handling," August 24, 2020. Accessed January 16, 2022. https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp.

Compilers typically provide a setting that enables you to disable assertions without having to modify your code. For example, `g++` allows the command-line option `-DNDEBUG`.

12.6.2 Failing Fast

CG The C++ Core Guidelines suggest that "If you can't throw exceptions, consider failing fast."[34] **Fail-fast** is a development style in which, rather than catching an exception, processing it and leaving your program in a state where it might fail later, the program terminates immediately.[35] This seems counterintuitive. The idea is that failing fast actually helps you build more robust software by finding and fixing errors sooner during development. Fewer errors are likely to make their way into the final product.[36]

34. C++ Core Guidelines, "E.26: If You Can't Throw Exceptions, Consider Failing Fast." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-no-throw-crash.

35. "Fail-Fast." Wikipedia. Wikimedia Foundation. Accessed January 16, 2022. https://en.wikipedia.org/wiki/Fail-fast.

36. Jim Shore, "Fail Fast [Software Debugging]." *IEEE Software* 21, no. 5 (September/October 2004): 21–25. Edited by Martin Fowler. https://martinfowler.com/ieeeSoftware/failFast.pdf.

12.7 Constructors, Destructors and Exception Handling

There are some subtle issues regarding exceptions in the context of constructors and destructors. In this section, you'll see

- why constructors should throw exceptions when they encounter errors,
- how to catch exceptions that occur in a constructor's member initializers and
- why destructors should not throw exceptions.

12.7.1 Throwing Exceptions from Constructors

SE First, consider an issue we've mentioned but not yet resolved. What happens when an error is detected in a constructor? For example, how should an object's constructor respond when it receives invalid data? Because **the constructor cannot return a value to indicate an error**, we must somehow indicate that the object was not constructed properly. One scheme is to return the improperly constructed object and hope that anyone using it would make appropriate tests to determine that it's in an inconsistent state. Another is to set a variable outside the constructor, such as a global error variable, but that's considered poor software engineering. The preferred alternative is to require the constructor to `throw` an exception that contains the error information. However, do not throw exceptions from the constructor of a global object. Such exceptions cannot be caught because they're constructed *before* `main` executes.

SE SE A constructor should throw an exception if a problem occurs while initializing an object. If the constructor manages dynamically allocated memory without smart pointers, it should release that memory before throwing an exception to prevent memory leaks. The destructor will not be called to release resources, though destructors for data members that have already been constructed will be called. **Always use smart pointers to manage dynamically allocated memory.**

12.7.2 Catching Exceptions in Constructors via Function `try` Blocks

Recall that base-class initializers and member initializers execute *before* the constructor's body. So, if you want to catch exceptions thrown by those initializers, you cannot simply wrap a constructor's body statements in a `try` block. Instead, you must use a **function `try` block**, which we demonstrate in Fig. 12.5.

```
1  // fig12_05.cpp
2  // Demonstrating a function try block.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <limits>
6  #include <stdexcept>
7
8  // class Integer purposely throws an exception from its constructor
9  class Integer {
10 public:
11    explicit Integer(int i) : value{i} {
12       std::cout << fmt::format("Integer constructor: {}\n", value)
13          << "Purposely throwing exception from Integer constructor\n";
14       throw std::runtime_error("Integer constructor failed");
15    }
16 private:
```

```
17      int value{};
18   };
19
20   class ResourceManager {
21   public:
22      ResourceManager(int i) try : myInteger(i) {
23         std::cout << "ResourceManager constructor called\n";
24      }
25      catch (const std::runtime_error& ex) {
26         std::cout << fmt::format(
27            "Exception while constructing ResourceManager: ", ex.what())
28            << "\nAutomatically rethrowing the exception\n";
29      }
30   private:
31      Integer myInteger;
32   };
33
34   int main() {
35      try {
36          const ResourceManager resource{7};
37      }
38      catch (const std::runtime_error& ex) {
39         std::cout << fmt::format("Rethrown exception caught in main: {}\n",
40            ex.what());
41      }
42   }
```

```
Integer constructor: 7
Purposely throwing exception from Integer constructor
Exception while constructing ResourceManager: Integer constructor failed
Automatically rethrowing the exception
Rethrown exception caught in main: Integer constructor failed
```

Fig. 12.5 Demonstrating a function `try` block.

To help demonstrate a function `try` block, class `Integer` (lines 9–18) simulates failing to "acquire a resource" by purposely throwing an exception (line 14) from its constructor.
Class `ResourceManager` (lines 20–32) contains an object of class `Integer` (line 31), which will be initialized in the `ResourceManager` constructor's member-initializer list.

In a constructor, you define a function `try` block by placing the `try` keyword *after* the constructor's parameter list and *before* the colon (`:`) that introduces the member-initializer list (line 22). The member-initializer list is followed by the constructor's body (lines 22–24). Any exceptions that occur in the member-initializer list or in the constructor's body can be handled by `catch` blocks that follow the constructor's body—in this example, the `catch` block at lines 25–29. The flow of control in this example is as follows:

- Line 36 in `main` creates an object of our `ResourceManager` class, which calls the class's constructor.
- Line 22 in the constructor calls class `Integer`'s constructor (lines 11–15) to initialize the `myInteger` object, producing the first two lines of output. Line 14 purposely throws an exception so we can demonstrate the `ResourceManager` constructor's function `try` block. This terminates class `Integer`'s constructor and throws the exception back to the initializer in line 22, which is in the `Resource-Manager` constructor's function `try` block.
- The function `try` block, which also includes the constructor's body, terminates.
- The `ResourceManager` constructor's `catch` handler at lines 25–29 catches the exception and displays the next two lines of output.

- SE The primary purpose of a constructor's function `try` block is to enable you to do initial exception processing, such as logging the exception or throwing a different exception that's more appropriate for your code. **Your object cannot be fully constructed, so each `catch` handler that follows a function `try` block is required to either throw a new exception or rethrow the existing one—explicitly or implicitly.**[37] Our `catch` handler does not explicitly contain a `throw` statement, so it implicitly rethrows the exception. This terminates the `ResourceManager` constructor and throws the exception back to line 36 in `main`.

37. "Function-try-block." Accessed January 16, 2022. https://en.cppreference.com/w/cpp/language/function-try-block.

- Line 36 is in a `try` block, so that block terminates, and the `catch` handler in lines 38–41 handles the exception, displaying the last line of the output.

SE Function `try` blocks also may be used with other functions using the following syntax:

```
void myFunction() try {
   // do something
}
catch (const ExceptionType& ex) {
   // exception processing
}
```

However, for a regular function, a function `try` block does not provide any additional benefit over simply placing the entire `try`...`catch` sequence in the function's body, as in:

```
void myFunction() {
   try {
      // do something
   }
   catch (const ExceptionType& ex) {
      // exception processing
   }
}
```

12.7.3 Exceptions and Destructors: Revisiting `noexcept(false)`

11 As of C++11, the compiler implicitly declares all destructors `noexcept` unless

- you say otherwise by declaring a destructor `noexcept(false)` or
- a direct or indirect base class's destructor is declared `noexcept(false)`.

SE If your destructor calls functions that might throw exceptions, you should catch and handle them, even if that simply means logging the exception and terminating the program in a controlled manner.[38] During destruction of a derived-class object, if it's possible for a base-class destructor to throw an exception, you can use a function `try` block on your derived-class destructor to ensure that you have the opportunity to catch the exception.

38. C++ Core Guidelines, "C.36: A Destructor Must Not Fail." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rc-dtor-fail.

If an exception occurs during object construction, destructors may be called:

- If an exception is thrown before an object is fully constructed, destructors will be called for any member objects or base-class subobjects constructed so far.

- If an array of objects has been partially constructed when an exception occurs, the destructors for only the array's constructed objects will be called.

Stack unwinding is guaranteed to have been completed when a `catch` handler begins executing. **If a destructor invoked due to stack unwinding throws an exception, the program terminates.** According to the `isocpp.org` FAQ,[39] the choice to terminate is because C++ does not know whether to

39. "How Can I Handle a Destructor That Fails?" Accessed January 16, 2022. https://isocpp.org/wiki/faq/exceptions#dtors-shouldnt-throw.

- continue processing the exception that led to stack unwinding in the first place or
- process the new exception thrown from the destructor.

12.8 Processing new Failures

Section 11.4 demonstrated dynamically allocating memory with `new`. Then, Section 11.5 showed modern C++ memory management using **RAII (Resource Acquisition Is Initialization)**, class template `unique_ptr` and function template `make_unique`, which uses operator `new` "under the hood." If `new` fails to allocate the requested memory, it `throw`s a `bad_alloc` exception (defined in header `<new>`).

In this section, Figs. 12.6–12.7 present two examples of `new` failing. Each attempts to acquire large amounts of dynamically allocated memory:

- The first example demonstrates `new` throwing a `bad_alloc` exception.
- The second uses function `set_new_handler` to specify a function to call when `new` fails. **This technique is mainly used in legacy C++ code written before compilers supported throwing `bad_alloc` exceptions.**

SE When an exception is thrown from the constructor for an object created in a `new` expression, the dynamically allocated memory for that object is released.

Do Not Throw Exceptions While Holding a Raw Pointer to Dynamically Allocated Memory

CG Section 11.5 showed that a `unique_ptr` enables you to ensure that dynamically allocated memory is properly deallocated regardless of whether the `unique_ptr` goes out of scope due to the normal flow of control or an exception. **When managing dynamically allocated memory via old-style raw pointers (as might be the case in legacy C++ code), do not allow uncaught exceptions to occur before you release the memory.**[40] For example, suppose a function contained the following series of statements:

40. C++ Core Guidelines, "E.13: Never Throw While Being the Direct Owner of an Object." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-never-throw.

```
int* ptr{new int[100]}; // acquire dynamically allocated memory
processArray(ptr); // assume this function might throw an exception
delete[] ptr; // return the memory to the system
// ...
```

A **memory leak** would occur if `processArray` throws an exception—the code would not reach the `delete[]` statement. To prevent this leak, you'd have to catch the exception and delete the memory before allowing the exception to propagate back to the caller. Instead, you should manage memory with `unique_ptr`, as discussed in Chapter 11.

12.8.1 `new` Throwing `bad_alloc` on Failure

Figure 12.6 demonstrates `new` throwing `bad_alloc` when `new` fails to allocate memory. The `for` statement (lines 15–19) inside the `try` block iterates through the `array` of `unique_ptr` objects named `items` and allocates to each element an array of 500,000,000 `double`s. Our primary test computer has 32 GB of RAM and eight TB of disk space. We had to allocate enormous numbers of elements to force dynamic memory allocation to fail. We used a separate test machine with 16 GB of RAM and 256 GB of disk space to produce this program's output and still had to allocate five billion `double`s to induce a memory allocation failure. You might be able to specify fewer than 500,000,000 on your system. If `new` fails during a call to `make_unique` and throws a `bad_alloc` exception, the loop terminates. The program continues in line 21, where the `catch` handler catches and processes the exception. Lines 22–23 print `"Exception occurred:"`, followed by the message returned from function `what`. Typically, this is an implementation-defined exception-specific message, such as `"bad allocation"` or `"std::bad_alloc"`. The output shows that the program performed only 10 iterations of the loop before `new` failed and threw the `bad_alloc` exception. Your output might differ based on your system's physical memory, the disk space available for virtual memory on your system and the compiler you're using.

```
1  // fig12_06.cpp
2  // Demonstrating standard new throwing bad_alloc when memory
3  // cannot be allocated.
4  #include <array>
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <memory>
8  #include <new> // bad_alloc class is defined here
9
10 int main() {
11    std::array<std::unique_ptr<double[]>, 1000> items{};
12
13    // aim each unique_ptr at a big block of memory
14    try {
15       for (int i{0}; auto& item : items) {
16          item = std::make_unique<double[]>(500'000'000);
17          std::cout << fmt::format(
18             "items[{}] points to 500,000,000 doubles\n", i++);
19       }
20    }
21    catch (const std::bad_alloc& memoryAllocationException) {
22       std::cerr << fmt::format("Exception occurred: {}\n",
23          memoryAllocationException.what());
24    }
25 }
```

```
items[0] points to 500,000,000 doubles
items[1] points to 500,000,000 doubles
items[2] points to 500,000,000 doubles
items[3] points to 500,000,000 doubles
items[4] points to 500,000,000 doubles
items[5] points to 500,000,000 doubles
items[6] points to 500,000,000 doubles
items[7] points to 500,000,000 doubles
items[8] points to 500,000,000 doubles
items[9] points to 500,000,000 doubles
Exception occurred: bad allocation
```

Fig. 12.6 `new` throwing `bad_alloc` on failure.

12.8.2 `new` Returning nullptr on Failure

SE You should use the version of `new` that throws `bad_alloc` exceptions on failure. However, the C++ standard specifies you also can use an older version of `new` that returns `nullptr` upon failure. For this purpose, header `<new>` defines object `std::nothrow` (of type `nothrow_t`), which is used as follows:

```
std::unique_ptr<double[]> ptr{
   new(std::nothrow) double[500'000'000]};
```

You cannot use `make_unique` here because it uses the default version of `new`.

12.8.3 Handling `new` Failures Using Function `set_new_handler`

In **legacy C++ code**, you might encounter another feature for handling `new` failures—the `set_new_handler` function (header `<new>`). This function takes as its argument either

- a pointer to a function that takes no arguments and returns `void` or
- a lambda that takes no arguments and returns `void`.

The function or lambda is called if `new` fails. **Once `set_new_handler` registers a `new handler` in the program, operator `new` does not throw `bad_alloc` on failure. Instead, it delegates the error handling to the `new`-handler function.**

The `new`-handler function should perform one of the following tasks:

1. Make more memory available by deleting other dynamically allocated memory or telling the user to close other applications, then try allocating memory again.
2. Throw an exception of type `bad_alloc` (or a derived class of `bad_alloc`).
3. Call function `abort` or `exit` (both found in header `<cstdlib>`) to terminate the program.

The `abort` function terminates a program immediately, whereas `exit` executes destructors for global objects and local `static` objects before terminating the program. **Non-`static` local objects are not destroyed when either of these functions is called.**

Figure 12.7 demonstrates `set_new_handler` (line 21). Function `customNewHandler` (lines 11–14) prints an error message (line 12), then calls `exit` (line 13) to terminate the program. The constant `EXIT_FAILURE` is defined in the header `<cstdlib>`. The output shows that the loop iterated nine times before `new` failed and invoked function `customNew-Handler`. Your output might differ based on your compiler and the physical memory and disk space available for virtual memory on your system.

```
1  // fig12_07.cpp
2  // Demonstrating set_new_handler.
3  #include <array>
4  #include <cstdlib>
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <memory>
8  #include <new> // set_new_handler is defined here
9
10 // handle memory allocation failure
11 void customNewHandler() {
12    std::cerr << "customNewHandler was called\n";
13    std::exit(EXIT_FAILURE);
14 }
15
16 int main() {
17    std::array<std::unique_ptr<double[]>, 1000> items{};
18
19    // specify that customNewHandler should be called on
```

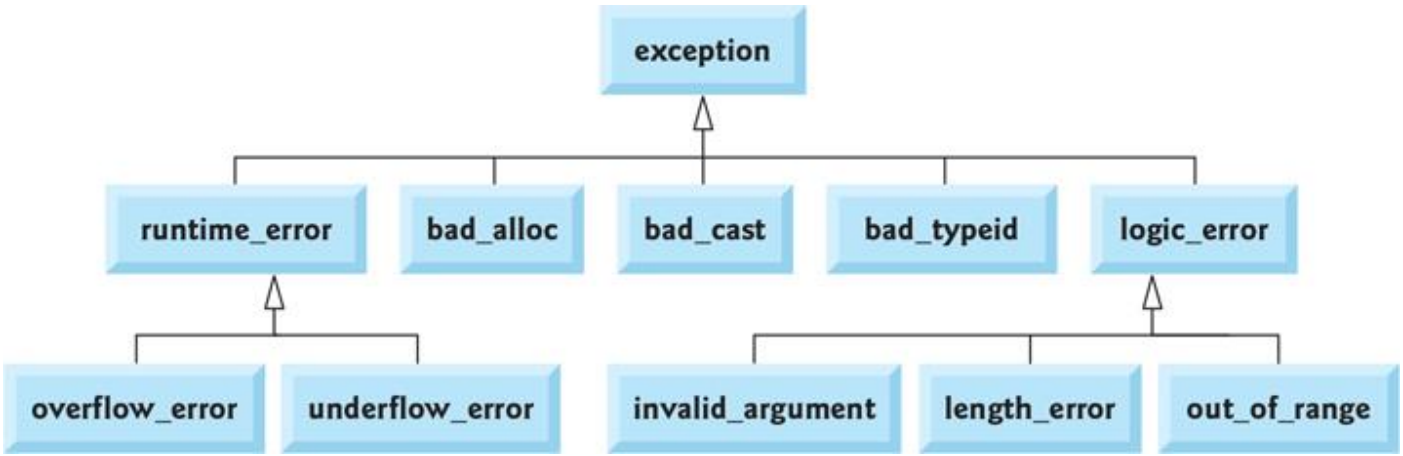
```
20      // memory allocation failure
21      std::set_new_handler(customNewHandler);
22
23      // aim each unique_ptr at a big block of memory
24      for (int i{0}; auto& item : items) {
25         item = std::make_unique<double[]>(500'000'000);
26         std::cout << fmt::format(
27             "items[{}] points to 500,000,000 doubles\n", i++);
28      }
29   }
```

```
items[0] points to 500,000,000 doubles
items[1] points to 500,000,000 doubles
items[2] points to 500,000,000 doubles
items[3] points to 500,000,000 doubles
items[4] points to 500,000,000 doubles
items[5] points to 500,000,000 doubles
items[6] points to 500,000,000 doubles
items[7] points to 500,000,000 doubles
items[8] points to 500,000,000 doubles
customNewHandler was called
```

Fig. 12.7 `set_new_handler` specifying the function to call when `new` fails.

12.9 Standard Library Exception Hierarchy

Exceptions fall nicely into several categories. The C++ standard library includes a hierarchy of exception classes, some of which are shown in the following diagram:



20 For a list of C++ standard library exception types—including the new C++20 exception types `nonexistent_local_time`, `ambiguous_local_time` and `format_error`—see

```
https://en.cppreference.com/w/cpp/error/exception
```

You can build programs that can `throw`

- standard exceptions,
- exceptions derived from the standard exceptions,
- your own exceptions not derived from the standard exceptions or
- instances of non-class types, like fundamental-type values and pointers.

SE CG The `exception` class hierarchy is a good starting point for creating custom exception types. In fact, rather than throwing exceptions of the types shown in the preceding diagram, **the C++**

Core Guidelines recommend creating derived-class exception types that are specific to your application. Such custom types can convey more meaning than the generically named exception types, like `runtime_error`.

Base Class `exception`

CG The standard library exception hierarchy is headed by base-class `exception` (defined in header `<exception>`). This class contains `virtual` function `what` that derived classes can override to issue an appropriate error message. If a `catch` handler specifies a reference to a base-class exception type, it can `catch` objects of all exception classes derived publicly from that base class.[41]

41. C++ Core Guidelines, "E.15: Catch Exceptions from a Hierarchy By Reference." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re-exception-ref.

Derived Classes of `exception`

Immediate derived classes of `exception` include `runtime_error` and `logic_error` (both defined in header `<stdexcept>`), each of which has several derived classes. Also derived from `exception` are the exceptions thrown by C++ operators:

- `bad_alloc` is thrown by `new` (Section 12.8),
- `bad_cast` is thrown by `dynamic_cast` (online Chapter 20) and
- `bad_typeid` is thrown by `typeid` (online Chapter 20).

Derived Classes of `runtime_error`

Class `runtime_error`, which we used briefly in Sections 12.2 and 12.5, is the base class of several other standard exception classes that indicate execution-time errors:

- Class `overflow_error` describes an **arithmetic overflow error** (i.e., the result is greater than the largest positive number or less than the largest negative number that can be stored in a given numeric type).
- Class `underflow_error` describes an **arithmetic underflow error**. This is for "subnormal floating-point values."[42]

42. "Subnormal number." Wikipedia. Wikimedia Foundation. Accessed January 16, 2022. https://en.wikipedia.org/wiki/Subnormal_number.

Derived Classes of `logic_error`

Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic:

- We used class `invalid_argument` in *set* functions (starting in Chapter 9) to indicate when an attempt was made to set an invalid value. Proper coding can, of course, prevent invalid arguments from reaching a function.
- Class `length_error` indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object.
- Class `out_of_range` indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Catching Exception Types Related By Inheritance

SE Using inheritance with exceptions enables an exception handler to `catch` related errors with concise notation:

- Err One approach is to `catch` each derived-class exception type individually. This is error-prone—you could forget to test explicitly for one or more of the derived-class types.
- A more concise approach is to `catch` references to base-class exception objects.

It's a logic error if you place a base-class `catch` handler before one that catches one of that base class's derived types—compilers will issue a warning for this. The base-class `catch` matches all objects of classes derived publicly from that base class, so the derived-class `catch` will never execute.[43]

43. C++ Core Guidelines, "E.31: Properly Order Your catch-Clauses." Accessed January 16, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Re_catch/.

Catching All Exceptions

Err SE C++ exceptions need not derive from class `exception`, so catching type `exception` is not guaranteed to `catch` all exceptions a program could encounter. **You can use `catch(...)` to catch all exception types thrown in a `try` block.** There are weaknesses to this approach:

- The type of the caught exception is unknown.
- Also, without a named parameter, you cannot refer to the exception object inside the exception handler.

SE The `catch(...)` handler is primarily used to perform recovery that does not depend on the exception type, such as releasing common resources. **The exception can be rethrown to alert enclosing `catch` handlers.**

12.10 C++'s Alternative to the finally Block: Resource Acquisition Is Initialization (RAII)

In several programming languages created after C++—such as Java, C# and Python—the `try` statement has an optional `finally` block that is guaranteed to execute, regardless of whether the corresponding `try` block completes successfully or terminates due to an exception. The `finally` block is placed after a `try` block's last exception handler or immediately after the `try` block if there are no exception handlers (which is allowed in those languages but not in C++). This makes `finally` blocks in those other languages a good mechanism for guaranteeing resource deallocation to prevent resource leaks.

As a programmer in another language, you might wonder why C++'s `try` statement has not added a `finally` block. In C++, we do not need `finally` due to **RAII (Resource Acquisition Is Initialization)**, smart pointers and destructors. If you design a class to use RAII, objects of your class will acquire their resources during object construction and deallocate them during object destruction.

Over the years, similar capabilities have been added to Java, C# and Python as well:

- Java has `try`-with-resources statements.
- C# has `using` statements.
- Python has `with` statements.

As program control enters these statements, each creates objects that acquire resources, which you can then use in the statements' bodies. When these statements terminate—successfully or due to an exception—they deallocate the resources automatically.

12.11 Some Libraries Support Both Exceptions and Error Codes

Exceptions are not universally used. A 2018 ISO worldwide C++ developer survey showed that exceptions were partially or fully banned in 52% of projects.[44] For example,

44. “C++ Developer Survey ‘Lite’: 2018–02.” February 2018. Accessed January 16, 2022. https://isocpp.org/files/papers/CppDevSurvey-2018-02-summary.pdf.

- the **Google C++ Style Guide**[45] indicates that Google does not use exceptions and

45. “Google C++ Style Guide.” Accessed January 16, 2022. https://google.github.io/styleguide/cppguide.html.

- the **Joint Strike Fighter Air Vehicle (JSF AV) C++ Coding Standards**[46] explicitly forbid using `try`, `catch` and `throw`.

46. “Joint Strike Fighter Air Vehicle C++ Coding Standards,” December 2005. Accessed January 16, 2022. https://www.stroustrup.com/JSF-AV-rules.pdf.

When organizations prohibit exceptions, they also prohibit using libraries with functions that might throw exceptions—such as the C++ standard library.

There are various disadvantages to not allowing exceptions in projects. Some problems one developer encountered in a project that banned exceptions included[47]

47. Lucian Radu Teodorescu’s answer to “Why do some people recommend not using exception handling in C++? Is this just a ‘culture’ in C++ community, or do some real reasons exist behind this?” August 2, 2015. Accessed January 16, 2022. https://www.quora.com/Why-do-some-people-recommend-not-using-exception-handling-in-C++-Is-this-just-a-culture-in-C++-community-or-do-some-real-reasons-exist-behind-this/answer/Lucian-Radu-Teodorescu.

- interoperability issues with class libraries that use exceptions,
- the amount of code required to deal with error conditions,
- problems with errors during object construction,
- problems with overloaded assignment operators,
- difficulties with error-handling flows of control,
- cluttering of the code by intermixing of logic and error handling,
- issues with resource allocation and deallocation and
- efficiency of the code.

To give programmers the flexibility of choosing whether to use exceptions, some libraries support dual interfaces with two versions of each function:

- one that throws an exception when it encounters a problem and
- one that sets or returns an error indicator when it encounters a problem.

17 As an example, consider the functions in C++17’s `<filesystem>` library.[48] These functions enable you to manipulate files and folders from C++ applications. In this library, each function has two versions—one that throws a `filesystem_error` exception and one that sets a value in its `error_code` argument that you pass to the function by reference.

48. “Filesystem Library.” Accessed January 16, 2022. https://en.cppreference.com/w/cpp/filesystem.

12.12 Logging

One common task when handling exceptions is to log where they occurred into a human-readable text file that developers can analyze later for debugging purposes. Logging can be used during development time to help locate and fix problems. It also can be used once an application ships—if an application crashes, it might write a log file that the user can then send to the developer.

Logging is not built into the C++ standard library, though you could create your own logging mechanisms using C++'s file-processing capabilities. However, there are many open-source C++ logging libraries:

- Boost.Log—https://www.boost.org/doc/libs/1_75_0/libs/log/doc/html/.
- Easylogging++—https://github.com/amrayn/easyloggingpp.
- Google Logging Library (glog)—https://github.com/google/glog.
- Loguru—https://github.com/emilk/loguru.
- Plog—https://github.com/SergiusTheBest/plog.
- spdlog—https://github.com/gabime/spdlog.

Some of these are header-only libraries that you can simply include in your projects. Others require installation procedures.

12.13 Looking Ahead to Contracts[49]

49. **Contracts are not yet a standard C++ feature**. The syntax we show here could change.

To strengthen your program's error-handling architecture, you can specify the expected states before and after a function's execution with preconditions and postconditions, respectively. We'll define each, show precondition code examples and explain what happens when contracts are violated

- A **precondition**[50] must be true when a function is invoked. Preconditions describe constraints on function parameters and any other expectations the function has just before it begins executing. **If the preconditions are not met, then the function's behavior is undefined**—it may throw an exception, proceed with an illegal value or attempt to recover from the error. If code with undefined behavior is allowed to proceed, the results could be unpredictable and not portable across platforms. Each function can have multiple preconditions.

50. "Precondition." Wikipedia. Wikimedia Foundation. Accessed January 16, 2022. https://en.wikipedia.org/wiki/Precondition.

- A **postcondition**[51] is true after the function successfully returns. Postconditions describe guarantees about the return value or side effects the function may have. When defining a function, you should document all postconditions so that others know what to expect when they call your function. You also should ensure that your function honors its postconditions if its preconditions are met. Each function can have multiple postconditions.

51. "Postcondition." Wikipedia. Wikimedia Foundation. Accessed January 16, 2022. https://en.wikipedia.org/wiki/Postcondition.

Precondition and Postcondition Violations

Today, precondition and postcondition violations often are dealt with by throwing exceptions. Consider `array` and `vector` function `at`, which receives an index into the container. For a precondition, function `at` requires that its index argument be greater than or equal to `0` and less than the container's size. If the precondition is met, `at`'s postcondition states that the function will return the item at that index; otherwise, `at` throws an `out_of_range` exception. As a client of an `array` or `vector`, we trust that function `at` satisfies its postcondition, provided that we meet the precondition.

Preconditions and postconditions are assertions, so you can implement them with the `assert` preprocessor macro (Section 12.6.1) as program control enters or exits a function. Preprocessor macros are generally deprecated in C++. Until contracts become part of C++, you'll

continue using the `assert` macro or custom C++ code to express preconditions, assertions and postconditions.

Invariants

An **invariant** is a condition that should always be true in your code—that is, a condition that never changes. **Class invariants** must be true for each object of a class. They generally are tied to an object's lifecycle. Class invariants remain true from the time an object is constructed until it's destructed. For example:

- Section 9.6's `Account` class requires that its `m_balance` data member always be non-negative.
- Section 9.7's `Time` class requires that its `m_hour` data member always have a value in the range 0 through 23, and its `m_minute` and `m_second` members always have values in the range 0 through 59.

These invariants ensure that objects of these classes always maintain valid state information throughout their lifetimes.

Functions also may contain invariants. For example, in a function that searches for a specified value in a `vector<int>`, the invariant is that if the value is in the `vector`, the value's index must be greater than or equal to 0 and less than the `vector`'s size.

Design By Contract

Design by contract (DbC)[52,53,54] is a **software-design approach** created by **Bertrand Meyer in the 1980s and used in the design of his Eiffel programming language**. Using this approach,

52. Bertrand Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1988.
53. Bertrand Meyer, In *Touch of Class: Learning to Program Well with Objects and Contracts*, xvii. Springer Berlin AN, 2016.
54. "Design by Contract." Wikipedia. Wikimedia Foundation. Accessed January 16, 2022. https://en.wikipedia.org/wiki/Design_by_contract.

- a function expects client code to meet the function's precondition(s),
- if the preconditions are true, the function guarantees its postcondition(s) will be true, and
- any invariants are maintained.

23 A proposal to add support for contract-based programming (commonly referred to as "contracts") to the C++ standard was first proposed in 2012 and later rejected.[55] Another proposal was eventually accepted for inclusion in C++20 but removed[56] late in C++20's development cycle due to "lingering design disagreements and concerns."[57] So, contracts have been pushed to at least C++23.

55. Nathan Meyers, "What Happened to C++20 Contracts?" August 5, 2019. Accessed January 16, 2022. https://www.reddit.com/r/cpp/comments/cmk7ek/what_happened_to_c20_contracts/.
56. Meyers, "What Happened to C++20 Contracts?"
57. Herb Sutter, "Trip Report: Summer ISO C++ Standards Meeting (Cologne)," July 2019. Accessed January 16, 2022. https://herbsutter.com/2019/07/20/trip-report-summer-iso-c-standards-meeting-cologne/.

Gradually Moving to Contracts in the C++ Standard Library

Herb Sutter says, "In Java and .NET, some 90% of all exceptions are thrown for precondition violations." He also says, "The programming world now broadly recognizes that programming bugs (e.g., out-of-bounds access, null dereference, and in general all pre/post/assert-condition violations) cause a corrupted state that cannot be recovered from programmatically, and so they should never be reported to the calling code as exceptions or error codes that code could somehow handle."[58]

58. Herb Sutter, “Trip Report: Summer Iso C++ Standards Meeting (Rapperswil),” July 2018. Accessed January 16, 2022. https://herbsutter.com/2018/07/.

SE **A key idea behind incorporating contracts is that many errors we currently deal with via exceptions can be located via preconditions and postconditions, then eliminated by fixing the code.**[59]

59. Glennen Carnie, “Contract Killing (in Modern C++),” September 18, 2019. Accessed January 16, 2022. https://blog.feabhas.com/2019/09/contract-killing-in-modern-c/.

Perf **A goal of contracts is to make most functions `noexcept`,**[60] which might enable the compiler to perform additional optimizations. Sutter says, **“Gradually switching precondition violations from exceptions to contracts promises to eventually remove a majority of all exceptions thrown by the standard library.”**[61,62]

60. Sutter, “Trip report: Summer ISO C++ standards meeting (Rapperswil).”

61. Sutter, “Trip report: Summer ISO C++ standards meeting (Rapperswil).”

62. To get a sense of the number of exceptions thrown by C++ and its libraries, we searched for the word “throws” in the final draft of the C++ standard document located at https://isocpp.org/files/papers/N4860.pdf. The document is over 1,800 pages—450+ pages cover the language, 1,000+cover the standard library and the rest are appendices, bibliography, cross references and indexes. “Throws” appears 422 times—there were 92 occurrences of “throws nothing,” 13 occurrences of “throws nothing unless...” (indicating an exception that is thrown as a result of stack unwinding) and 329 occurrences of functions that throw exceptions. Many of these 329 cases are examples of where contracts will help eliminate the need to throw exceptions.

Contracts Attributes

The contracts proposal[63] introduces three attributes of the form

63. G. Dos Reis, J. D. Garcia, J. Lakos, A. Meredith, N. Meyers and B. Stroustrup, “Support for Contract Based Programming in C++,” June, 8, 2018. Accessed January 16, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0542r5.html.

`[[`*contractAttribute optionalLevel optionalIdentifier: condition*`]]`

that you can use to specify preconditions, postconditions and assertions for your functions. The *optionalIdentifier* is one of the function’s local variables for use in postconditions, as you’ll see in Fig. 12.9. The contract attributes are

- `expects`—for specifying a function’s preconditions that are checked before the function’s body begins executing,
- `ensures`—for specifying a function’s postconditions that are checked just before the function returns and
- `assert`—for specifying assertions that are checked as they’re encountered throughout a function’s execution.

If you specify multiple preconditions and postconditions, they’re checked in their order of declaration.

Contracts Levels

There are three contract levels:

- `default` specifies a contract with little runtime overhead compared to the function's typical execution time. If a level is not specified, the compiler assumes `default`.
- Perf SE `audit` specifies a contract with **significant runtime overhead** compared to the function's typical execution time. Such contracts are intended primarily for use during program development.
- SE `axiom` specifies a contract meant to be enforced by static code checkers rather than at runtime.

Perf Using these levels will enable you to select which contracts are enforced at runtime, thus controlling the performance overhead. You can choose—presumably via compiler flags—whether to turn contracts off entirely, perform the low-overhead `default` contracts or perform the high-overhead `audit` contracts.

Specifying Preconditions, Postconditions and Assertions

Precondition contracts (`expects`) and postcondition contracts (`ensures`) are specified in a function's prototype—they are listed after the function's signature and before the semicolon. For example, a function that calculates the real (not imaginary) square root of a `double` value expects its argument to be greater than or equal to zero. You can specify this with an `expects` precondition contract:

```
double squareRoot(double value)
    [[expects: value >= 0.0]];
```

If the function definition also serves as the function prototype, the precondition and postcondition contracts are listed between the function's signature and its opening left brace.

Assertions are specified as statements in the function body. For instance, to assert that an integer exam `grade` is in the range `0` through `100`, you'd write

```
[[assert: grade >= 0 && grade <= 100]];
```

Note that the `assert` in the preceding statement is not an `assert` macro.

Early-Access Implementation

There is an early-access contracts implementation in GNU C++[64] that you can test through the **Compiler Explorer website**[65] (https://godbolt.org), which supports many compiler versions across various programming languages. You can choose "**x86-64 gcc (contracts)**" as your compiler and compile contracts-based code using the compiler options described at

```
https://gitlab.com/lock3/gcc-new/-/wikis/contract-assertions
```

64. There is also an early-access Clang contracts implementation at https://github.com/arcosuc3m/clang-contracts, but you need to build and install it yourself.

65. Copyright © 2012–2019, Compiler Explorer Authors. All rights reserved. Compiler Explorer is by Matt Godbolt. https://xania.org/MattGodbolt.

We provide a `godbolt.org` URL where you can try each of our examples using the early-access implementation. The GNU C++ early-access contracts implementation uses different keywords for preconditions and postconditions:

- `pre` rather than `expects` and
- `post` rather than `ensures`.

Example: Division-By-Zero

Figure 12.8 reimplements our `quotient` function from Fig. 12.3. Here, we specify in the `quotient` function's prototype (lines 5–6) a `default` level precondition contract indicating that the `denominator` must not be `0.0`. The `default` keyword also can be specified explicitly, as in

```
[[pre default: denominator != 0.0]]
```

```
1  // fig12_08.cpp
2  // quotient function with a contract precondition.
3  #include <iostream>
4
5  double quotient(double numerator, double denominator)
6     [[pre: denominator != 0.0]];
7
8  int main() {
9     std::cout << "quotient(100, 7): " << quotient(100, 7)
10       << "\nquotient(100, 0): " << quotient(100, 0) << '\n';
11 }
12
13 // perform division
14 double quotient(double numerator, double denominator) {
15    return numerator / denominator;
16 }
```

Fig. 12.8 quotient function with a contract precondition.

This example can be found at

```
https://godbolt.org/z/jn4MK3o9T
```

As you type code into the Compiler Explorer editor, or when you load an existing example, Compiler Explorer automatically compiles and runs it or displays any compilation errors.

We preset the compiler options for this example to

```
-std=c++20 -fcontracts
```

20 which compiles the code with C++20 and experimental contracts support. Compiler Explorer generally shows at least two tabs—a code editor and a compiler. Our examples also show the output tab so you can see the executed program's results—this is also where compilation errors would be displayed. The compiler tab has two drop-down lists at the top. One lets you select the compiler. The other lets you view and edit the compiler options, or you can click the down arrow to select common compiler options.

The `quotient` call at line 9 satisfies the precondition and executes successfully, producing

```
quotient(100, 7): 14.2857
```

The `quotient` call at line 10, however, causes a **contract violation**. So, the **default violation handler** (`handle_contract_violation`) is implicitly called, displays the following error message, then terminates the program:

```
default std::handle_contract_violation called:
 ./example.cpp 6 quotient denominator != 0.0 default default 0
```

Changing the compilation options to

```
-std=c++20 -fcontracts -fcontract-build-level=off
```

disables contract checking and allows line 10 to execute, producing the output

```
quotient(100, 0): inf
```

Division-by-zero is undefined behavior in C++ but many compilers, including the three key compilers we use throughout this book, return positive infinity (`inf`) or negative infinity (`-inf`) in this case. This is the behavior specified by the **IEEE 754 standard for floating-point arithmetic**, which is widely supported by modern programming languages.

Contract Continuation Mode

The default **continuation mode** for contract violations is to terminate the program immediately. To allow a program to continue executing, you can add the compiler option

```
-fcontract-continuation-mode=on
```

Example: Search Function Requiring a Sorted `vector`

Consider Fig. 12.9, which defines a `binarySearch` function template with two preconditions—the `vector` argument must contain elements and must be sorted. You can test the example at

```
https://godbolt.org/z/obMsY5Wo4
```

```
1  // fig12_09.cpp
2  // binarySearch function with a precondition requiring a sorted vector.
3  #include <algorithm>
4  #include <iostream>
5  #include <vector>
6
7  template<typename T>
8  int binarySearch(const std::vector<T>& items, const T& key)
9     [[pre: items.size() > 0]]
```

```
10      [[pre audit: std::is_sorted(items.begin(), items.end())]] {
11      size_t low{0}; // low index of elements to search
12      size_t high{items.size() - 1}; // high index
13      size_t middle{(low + high + 1) / 2}; // middle element
14      int loc{-1}; // key's index; -1 if not found
15
16      do {// loop to search for element
17         // if the element is found at the middle
18         if (key == items[middle]) {
19            loc = middle; // loc is the current middle
20         }
21         else if (key < items[middle]) { // middle is too high
22            high = middle - 1; // eliminate the higher half
23         }
24         else { // middle element is too low
25            low = middle + 1; // eliminate the lower half
26         }
27
28         middle = (low + high + 1) / 2; // recalculate the middle
29      } while ((low <= high) && (loc == -1));
30
31      return loc; // return location of key
32   }
33
34   int main() {
35      // sorted vector v1 satisfies binarySearch's sorted vector precondition
36      std::vector v1{10, 20, 30, 40, 50, 60, 70, 80, 90};
37      int result1{binarySearch(v1, 70)};
38      std::cout << "70 was " << (result1 != -1 ? "" : "not ")
39         << "found in v1\n";
40
41      // unsorted vector v2 violates binarySearch's sorted vector precondition
42      std::vector v2{60, 70, 80, 90, 10, 20, 30, 40, 50};
43      int result2{binarySearch(v2, 60)};
44      std::cout << "60 was " << (result2 != -1 ? "" : "not ")
45         << "found in v2\n";
46   }
```

Fig. 12.9 `binarySearch` function with a precondition and a postcondition.

In this example, we defined the function before `main` and placed the preconditions after the parameter list and before the function's body.

The `binarySearch` function template performs a binary search on the `vector`, which requires the `vector` to be in sorted order; otherwise, the result could be incorrect. Our implementation also requires the `vector` to contain elements. So we declared the preconditions

```
[[pre: items.size() > 0]]
[[pre audit: is_sorted(begin(items), end(items))]]
```

The first ensures that the vector contains elements. The second calls the C++ standard library function `is_sorted` (from header `<algorithm>`[66]) to check whether the `vector` is sorted. This is potentially an expensive operation. A binary search of a billion-element sorted `vector` requires only 30 comparisons. However, determining whether the `vector` is sorted requires 999,999,999 comparisons. And sorting one billion elements efficiently with an $O(n \log n)$ sort algorithm could require about 30

billion operations. A developer might want to enable this test during development and disable it in production code. For this reason, we specified the precondition contract level `audit`.
66. There are over 200 functions in the C++ standard library's <algorithm> header. We survey many of these in Chapter 14.

We preset the compiler options for this example to

```
-std=c++2a -fcontracts
```

20 which, again, compiles the code with C++20 and `default` level contracts support, so **the line 10 `audit` level precondition contract is ignored**. In `main`, we created two `vector`s of integers containing the same values—`v1` is sorted (line 36) and `v2` is unsorted (line 42). With the initial compiler settings, the precondition that ensures the `vector` is sorted is not tested, sothe `binarySearch` calls in lines 16 and 21 complete, and the program displays the output:

```
70 was found in v1
60 was not found in v2
```

Because the **`audit` level** precondition was ignored, our program has a logic error. The second line of output shows that `60` was not found in `v2`. Again, the algorithm expects `v2` to be sorted, so it failed to find `60`, even though it's in `v2`. Changing the **contract build level** to `audit` in the compilation options, as in

```
-std=c++2a -fcontracts -fcontract-build-level=audit
```

enables `audit` level contract checking. When the program runs with `audit` contracts enabled, the `vector` `v1` in line 37's `binarySearch` call satisfies the precondition in line 10 and, as before, lines 38–39 output

```
70 was found in v1
```

However, the `vector v2` in line 43's `binarySearch` call causes a **contract violation**, resulting in the error message:

```
default std::handle_contract_violation called:
 ./example.cpp 10 binarySearch<int> is_sorted(begin(items), end(items)) audit
default 0
```

Custom Contract Violation Handler

The examples so far used the default contract violation handler. When a contract violation occurs, a `contract_violation` object is created containing the following information:

- the line number of the violation—returned by member function `line_number`,
- the source-code filename—returned by member function `file_name`,
- the function name in which the violation occurred—returned by member function `function_name`,
- a description of the condition that was violated—returned by member function `comment` and
- the contract level—returned by member function `assertion_level`.

The `contract_violation` is passed to the **violation handler** function of the form

```
void handle_contract_violation(const contract_violation& violation) {
```

```
   // handler code
}
```

The experimental contracts implementation in `g++` provides a default implementation of this function. If you define your own, `g++` will call your version.

12.14 Wrap-Up

C++ is used to build real-world, mission-critical and business-critical software. The systems it's used for are often massive. In this chapter, you learned that it's essential to eliminate bugs during development and decide how to handle problems once the software is in production.

We discussed the ways that exceptions may surface in your code. We discussed how exception handling helps you write robust, fault-tolerant programs that catch infrequent problems and continue executing, perform appropriate cleanup and terminate gracefully, or terminate abruptly in the case of unanticipated exceptions.

We reviewed exception-handling concepts in an example that demonstrated the flows of control when a program executes successfully and when an exception occurs. We discussed use-cases for catching, then rethrowing exceptions. We showed how stack unwinding enables other functions to handle exceptions that are not caught in a particular scope.

You learned when to use exceptions. We also introduced the exception guarantees you can provide in your code—no guarantee, the basic exception guarantee, the strong exception guarantee and the no-throw exception guarantee. We discussed why exceptions are used to indicate errors during construction and why destructors should not throw exceptions. We also showed how to use function `try` blocks to catch exceptions from a constructor's member-initializer list or from base-class destructors when a derived-class object is destroyed.

You saw that operator `new` throws `bad_alloc` exceptions when dynamic memory allocation fails. We also showed how dynamic memory allocation failures were handled in legacy C++ code with `set_new_handler`.

We introduced the C++ standard library exception class hierarchy and created a custom exception class that inherited from a C++ standard library exception class. You learned why it's important to catch exceptions by reference to enable exception handlers to catch exception types related by inheritance and without slicing. We also introduced logging exceptions into a file that developers can analyze later for debugging purposes.

We discussed why some organizations disallow exception handling. You also saw that some libraries provide dual interfaces, so developers can choose whether to use versions of functions that throw exceptions or versions that set error codes.

The chapter concluded with an introduction to the contracts feature, originally adopted for C++20 but delayed to a future C++ version. We showed how to use contracts to test preconditions, postconditions and assertions at runtime. You learned that these test conditions that should always be true in correct code. You saw that if such conditions are false, contract violations occur and, by default, the code terminates immediately. This enables you to find errors faster, eliminate them during development and create more robust code.

Chapter 6 introduced the `array` and `vector` standard library classes. In Chapter 13, you'll learn about many additional C++ standard library containers as well as iterators, which are used by standard library algorithms to walk through containers and manipulate their elements.

13. Standard Library Containers and Iterators

Objectives

In this chapter, you'll:

- Learn more about the C++ standard library's reusable containers, iterators and algorithms.
- Understand how containers relate to C++20 ranges.
- Use I/O stream iterators to read values from the standard input stream and write values to the standard output stream.
- Use iterators to access container elements.
- Use the `vector`, `list` and `deque` sequence containers.
- Use `ostream_iterator`s with the `std::copy` and `std::ranges::copy` algorithms to output container elements in a single statement.
- Use the `set`, `multiset`, `map` and `multimap` ordered associative containers.
- Understand the differences between the ordered and unordered associative containers.
- Use the `stack`, `queue` and `priority_queue` container adaptors.
- Use the `bitset` "near container" to manipulate a collection of bit flags.

Outline

13.1 Introduction
13.2 Introduction to Containers
13.2.1 Common Nested Types in Sequence and Associative Containers
13.2.2 Common Container Member and Non-Member Functions
13.2.3 Requirements for Container Elements
13.3 Working with Iterators
13.3.1 Using `istream_iterator` for Input and `ostream_iterator` for Output
13.3.2 Iterator Categories
13.3.3 Container Support for Iterators
13.3.4 Predefined Iterator Type Names
13.3.5 Iterator Operators
13.4 A Brief Introduction to Algorithms
13.5 Sequence Containers
13.6 `vector` Sequence Container
13.6.1 Using `vector`s and Iterators
13.6.2 `vector` Element-Manipulation Functions
13.7 `list` Sequence Container
13.8 `deque` Sequence Container
13.9 Associative Containers
13.9.1 `multiset` Associative Container
13.9.2 `set` Associative Container
13.9.3 `multimap` Associative Container
13.9.4 `map` Associative Container
13.10 Container Adaptors
13.10.1 `stack` Adaptor
13.10.2 `queue` Adaptor
13.10.3 `priority_queue` Adaptor

13.11 `bitset` Near Container
13.12 Optional: A Brief Intro to Big *O*
13.13 Optional: A Brief Intro to Hash Tables
13.14 Wrap-Up

13.1 Introduction

The standard library defines powerful, template-based, reusable components that implement many common data structures and algorithms used to process those data structures. We began introducing templates in Chapters 5–6 and use them extensively here and in Chapters 14 and 15. Historically, the features presented in this chapter were referred to as the **Standard Template Library** or **STL**.[1] In the C++ standard document, they are simply referred to as part of the C++ standard library.

---

1. The STL was developed by Alexander Stepanov and Meng Lee at Hewlett-Packard and is based on their generic programming research, with significant contributions from David Musser. Stepanov first proposed the STL for inclusion in C++ at the November 1993 ANSI/ISO C++ standardization committee meeting. It was approved for inclusion in July 1994 (https://en.wikipedia.org/wiki/History_of_the_Standard_Template_Library; accessed January 22, 2022).

Containers, Iterators and Algorithms

This chapter introduces three key standard library components—**containers** (templatized data structures), iterators and algorithms. We'll introduce **containers**, **container adaptors** and **near containers**.

Common Member Functions Among Containers

Each container has associated member functions—a subset of these is defined in all containers. We illustrate most of this common functionality in our examples of `array` (introduced in Chapter 6), `vector` (also introduced in Chapter 6 and covered in more depth here), `list` (Section 13.7) and `deque` (pronounced "deck"; Section 13.8).

Iterators

Iterators, which have properties similar to those of **pointers**, are used to manipulate container elements. **Built-in arrays** also can be manipulated by standard library algorithms, using pointers as iterators. We'll see that manipulating containers via iterators provides tremendous expressive power when combined with standard library algorithms—sometimes reducing many lines of code to a single statement.

Algorithms

Standard library **algorithms** (which we'll cover in-depth in Chapter 14) are function templates that perform common data manipulations, such as **searching**, **sorting**, **copying**, **transforming** and **comparing elements or entire containers**. The standard library provides scores of algorithms:

- 20 90 in the `<algorithm>` header's `std` namespace—82 also are overloaded in the `std::ranges` namespace for use with C++20 ranges,
- 11 in the `<numeric>` header's `std` namespace,
- 20 14 in the `<memory>` header's `std` namespace—all 14 also are overloaded in the `std::ranges` namespace for use with C++20 ranges and
- 2 in the `<cstdlib>` header.

Many were added in C++11 and C++20, and a few in C++17. For the complete list of algorithms with links to their descriptions, visit:

```
https://en.cppreference.com/w/cpp/algorithm
```

Most algorithms use iterators to access container elements. Each algorithm has minimum requirements for the kinds of iterators that can be used with it. We'll see that containers support specific kinds of iterators, some more powerful than others. The iterators a container supports determine whether the container can be used with a specific algorithm. Iterators encapsulate the mechanisms used to traverse containers and access their elements, enabling many algorithms to be applied to containers with significantly different implementations. This also enables you to create new algorithms that can process the elements of multiple container types.

C++20 Ranges

20 Chapter 6 introduced C++20's new **ranges** and **views**. You saw that a **range** is a collection of elements you can iterate over. So, `array`s and `vector`s are ranges. You also used **views** to specify **pipelines** of operations that manipulate ranges of elements. Any container with iterators representing its beginning and end can be treated as a C++20 range. In this chapter, we'll use the new C++20 standard library algorithm `std::ranges::copy` and the older C++ standard library algorithm `std::copy` to demonstrate how ranges simplify your code. In Chapter 14, we'll use many more C++20 algorithms from the `std::ranges` namespace to demonstrate additional **ranges** and **views** features.

Custom Templatized Data Structures

Some popular data structures include linked lists, queues, stacks and binary trees:

- Linked lists are collections of data items logically "lined up in a row."Insertions and removals are made anywhere in a linked list.
- **Stacks** are important in compilers and operating systems. Insertions and removals are made **only** at one end of a stack—its **top**.
- Queues represent waiting lines. Insertions are made at the back (also called the **tail**), and removals are made from the front (also called the **head**).
- Binary trees are nonlinear, hierarchical data structures that facilitate **searching** and **sorting** data and **duplicate elimination**.

Each of these data structures has many other interesting applications. We can carefully weave linked objects together with pointers, but pointer-based code is complex and can be error-prone. The slightest omissions or oversights can lead to serious **memory-access violations** and **memory leaks** with no forewarning from the compiler.

CG SE Perf Avoid reinventing the wheel. When possible, use the C++ standard library's preexisting containers, iterators and algorithms.[2] The prepackaged container classes provide the data structures you need for most applications. Using the standard library's proven containers, iterators and algorithms helps you reduce testing and debugging time. They were conceived and designed for performance and flexibility.

2. C++ Core Guidelines, "SL.1: Use Libraries Wherever Possible." Accessed January 22, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-lib.

13.2 Introduction to Containers[3]

3. This section is intended as an introduction to a reference-oriented chapter. You may want to read it quickly and refer back to it as necessary when reading the chapter's live-code examples.

The standard library containers are divided into four major categories:

- sequence containers,
- ordered associative containers,
- unordered associative containers and
- container adaptors.

We briefly summarize the containers here and show many live-code examples in the following sections.

Sequence Containers

The **sequence containers** represent linear data structures with all of their elements conceptually "lined up in a row," such as `array`s, `vector`s and linked lists, all of which are sortable. The five **sequence containers** are

- `array` (Chapter 6)—Fixed size. Elements are contiguous in memory. **Direct access (also called random access) to any element.**
- `vector` (Section 13.6)—Resizable. Elements are contiguous in memory. Rapid insertions and deletions at the back. Direct access to any element.
- `list` (Section 13.7)—Resizable doubly linked list, rapid insertion and deletion anywhere.
- `deque` (Section 13.8)—Resizable. Rapid insertions and deletions at the front or back. Direct access to any element.
- 11 `forward_list`—Resizable singly linked list, rapid insertion and deletion anywhere.

Linked-list containers do not support random access to any element. Class `string` supports the same functionality as a sequence container but stores only character data.

Associative Containers

Associative containers are nonlinear data structures (usually implemented as binary trees[4,5]) that typically can quickly locate elements. Such containers can store sets of values or **key–value pairs** in which each key has an associated value. For example, a program might associate employee IDs with `Employee` objects. As you'll see, some associative containers allow multiple values for each

key. **The keys in associative containers are immutable**—they cannot be modified unless you first remove them from the container. The four **ordered associative containers** are

4. "set." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/set.
5. "map." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/map.

- `multiset` (Section 13.9.1)—Rapid lookup, duplicates allowed.
- `set` (Section 13.9.2)—Rapid lookup, no duplicates allowed.
- `multimap` (Section 13.9.3)—Rapid key-based lookup, duplicate keys allowed. One-to-many mapping.
- `map` (Section 13.9.4)—Rapid key-based lookup, no duplicate keys allowed. Oneto-one mapping.

The four **unordered associative containers** (implemented using hashing[6,7]) are

6. "unordered_set." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/unordered_set.

7. "unordered_map." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/unordered_map.

- `unordered_multiset`—Rapid lookup, duplicates allowed.
- `unordered_set`—Rapid lookup, no duplicates allowed.
- `unordered_multimap`—One-to-many mapping, duplicates allowed, rapid key-based lookup.
- `unordered_map`—One-to-one mapping, no duplicates allowed, rapid key-based lookup.

Container Adaptors

The standard library implements `stack`, `queue` and `priority_queue` as container adaptors that enable a program to view a sequence container in a constrained manner. The three container adaptors are

- `stack`—Last-in, first-out (LIFO) data structure.
- `queue`—First-in, first-out (FIFO) data structure.
- `priority_queue`—Highest-priority element is always the first element out.

Near Containers

There are other container types that are considered near containers—built-in arrays (Chapter 7), `bitset`s (Section 13.11) for maintaining sets of flag values, `string`s and `valarray`s for performing high-speed mathematical vector operations[8] (not to be confused with the `vector` container). These types are considered near containers because they exhibit some, but not all, capabilities of the **sequence** and **associative containers**.

8. For overviews of valarray and its mathematical capabilities, see its documentation at https://en.cppreference.com/w/cpp/numeric/valarray and check out the article "std::valarray Class in C++" at https://www.geeksforgeeks.org/std-valarray-class-c/.

13.2.1 Common Nested Types in Sequence and Associative Containers

The following table shows the common **nested types** defined inside each sequence container and associative container class definition. These are used in template-based variables declarations, parameters to functions and return values from functions, as you'll see in this chapter and Chapter 14. For example, the `value_type` in each container always represents the container's element type.

Nested type	Description
`allocator_type`	The type of the object used to **allocate the container's memory**—not included in the `array` container. Containers that use allocators each provide a default allocator, which is sufficient for most programmers. Custom allocators are beyond this book's scope.
`value_type`	The type of the container's elements.
`reference`	The type used to declare a reference to a container element.
`const_reference`	The type used to declare a reference to a `const` container element.
`pointer`	The type used to declare a pointer to a container element.
`const_pointer`	The type used to declare a pointer to a `const` container element.
`iterator`	An iterator that points to a container element.
`const_iterator`	An iterator that points to an element of a `const` container. Used only to **read** elements and to perform `const` operations.
`reverse_iterator`	A reverse iterator that points to a container element. Iterates through a container **back-to-front**. Not provided for `forward_list`.
`const_reverse_iterator`	A reverse iterator that points to a container element and can be used only to **read** elements and to perform `const` **operations**. Used to iterate through a container in reverse. Not provided for `forward_list`.

Nested type	Description
`difference_type`	A type representing the number of elements between two iterators that refer to elements of the same container. A value of this type is returned by an iterator's overloaded minus (-) operator, if it has one.
`size_type`	The **type used to count items** in a container and **index** through a random-access sequence container.

13.2.2 Common Container Member and Non-Member Functions

Most containers provide similar functionality. Many operations apply to all containers, and others apply to specific container categories. The following tables describe the commonly available functions in most standard library containers. Before using any container, you should study its capabilities. For a complete list of all the container member functions and which containers support them, see the **`cppreference.com` member function table.**[9] Several member functions we do not list in this section are covered throughout the chapter as we present various sequence containers, associative containers and container adaptors.

9. "Container Library—Member Functions Table." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container.

Container Special Member Functions

The following table describes the container special member functions provided by each container. In addition, each container class typically provides many overloaded constructors for initializing containers and container adaptors in various ways. For example, each sequence and associative container can be initialized from an `initializer_list`.

Container special member function	Description
default constructor	A constructor that initializes an empty container.
copy constructor	A constructor that initializes the container to be a copy of an existing container of the same type.
copy `operator=`	Copies the elements of one container into another.
move constructor	Moves the contents of an existing container into a new one of the same type. The old container no longer contains the data. This avoids the overhead of copying each element of the existing container.

Container special member function	Description
move `operator=`	Moves the contents of one container into another of the same type. The old container no longer contains the data. This avoids the overhead of copying each element of the existing container.
destructor	Destructor function to destroy the container elements.

Non-Member Relational and Equality Operators

20 The `<`, `<=`, `>`, `>=`, `==` and `!=` operators are supported by most containers. In C++17 and earlier, these are overloaded as non-member functions. In C++20, the `<`, `<=`, `>`, `>=` and `!=` operators are synthesized by the compiler using the new **three-way comparison operator** `<=>` and the `==` operator. As we showed in Section 11.7, the C++ compiler can use `<=>` to implement each relational and equality comparison by rewriting it in terms of `<=>`. For example, the compiler rewrites `x < y` as

```
(x <=> y) < 0
```

The unordered associative containers do not support `<`, `<=`, `>` and `>=`. The relational and equality operators are not supported for `priority_queue`s.

Member Functions That Return Iterators

The following table shows container member functions that return iterators. The container **`forward_list`** does not have the member functions `rbegin`, `rend`, `crbegin` and `crend`.

Container member function	Description
`begin`	Returns an `iterator` or a `const_iterator` (depending on whether the container is `const`) referring to the container's **first element**.
`end`	Returns an `iterator` or a `const_iterator` (depending on whether the container is `const`) referring to the **next position after the end** of the container.
11 `cbegin` (C+ + 11)	Returns a `const_iterator` referring to the container's **first element**.
`cend` (C+ + 11)	Returns a `const_iterator` referring to the **next position after the end** of the container.
`rbegin`	Returns a `reverse_iterator` or a `const_reverse_iterator` (depending on whether the container is `const`) referring to the container's **last element**.

Container member function	Description
`rend`	Returns a `reverse_iterator` or a `const_reverse_iterator` (depending on whether the container is `const`) referring to the **position before the container's first element**.
11 `crbegin` (C++11)	Returns a `const_reverse_iterator` referring to the container's **last element**.
`crend` (C+ + 11)	Returns a `const_reverse_iterator` referring to the **position before the container's first element**.

Other Member Functions

The following table lists various additional container member functions. If a function is not supported for all containers, we specify which do or do not support it. For a container's complete list of member functions, see its documentation page, which you can access from

```
https://en.cppreference.com/w/cpp/container
```

Container member function	Description
`clear`	Removes all elements from the container. Not supported for `array`s.
20 `contains` (C++20)	Returns `true` if the specified key is present in the container; otherwise, returns `false`. Supported only for associative containers.
`empty`	Returns `true` if the container is empty; otherwise, returns `false`.
11 `emplace` (C++11)	Constructs an item in place where it will reside in the container. If there is already an item in that position, it constructs the object in a separate location, then moves it into place with move assignment. Not supported for `array`s. A `forward_list` supports `emplace_after` and `emplace_front`, not `erase`. The associative containers provide `emplace` and `emplace_hint`.
`erase`	Removes one or more elements from the container. Not supported for `array`s. A `forward_list` supports `erase_after`, not `erase`.
17 `extract` (C++17)	The associative containers are **linked data structures** of **nodes** containing values. Associative container

Container member function	Description
	member function `extract` removes a node from the container and returns an object of that container's `node_type`.
`insert`	Inserts an item in the container. This is overloaded to support **copy and move semantics**. Not supported for `array`s. A `forward_list` supports `insert_after`, not `insert`.
`max_size`	Returns the maximum number of elements for a container.
`size`	Returns the number of elements currently in the container. Not supported for `forward_list`s.
`swap`	Swaps the contents of two containers.

13.2.3 Requirements for Container Elements

Before using a standard library container, it's important to ensure that the element type supports a minimum set of functionality. For example:

- If inserting an item into a container requires a **copy** of the object, the object type should provide a **copy constructor** and **copy assignment operator**.
- If inserting an item into a container requires **moving** the object, the object type should provide a **move constructor** and **move assignment operator**—Chapter 11 discussed **move semantics**.
- The **ordered associative containers** and many algorithms require elements to be compared. For this reason, the object type should support comparisons.

As you review the documentation for each container, whether in the C++ standard document itself or on sites like cppreference.com, you'll see various **named requirements**, such as **CopyConstructible**, **MoveAssignable** or **EqualityComparable**. These help you determine whether your types are compatible with various C++ standard library containers and algorithms. You can view a list of named requirements and their descriptions at

```
https://en.cppreference.com/w/cpp/named_req
```

20 In C++20, many named requirements are formalized as **concepts**, which we'll discuss in Chapters 14 and 15.

13.3 Working with Iterators

Iterators have many similarities to pointers. They point to elements in sequence containers and associative containers, but they also hold state information sensitive to the particular containers on which they operate. So, iterators are implemented for each type of container. Iterator operation

syntax is uniform across containers. For example, applying `*` to an iterator dereferences it so that you can access the referenced element. Applying `++` to an iterator moves it to the container's next element. This uniformity is a key aspect of iterators, enabling standard library algorithms to operate on various container types using compile-time polymorphism.

Sequence containers and associative containers provide member functions `begin` and `end`. Function `begin` returns an iterator pointing to the container's first element. Function `end` returns an iterator pointing to the **first element past the end of the container** (one past the end)—a nonexistent element that's frequently used to determine when the end of a container is reached. You'll often use this in equality or inequality comparisons to determine whether an incremented iterator has reached the end of the container.

13.3.1 Using `istream_iterator` for Input and `ostream_iterator` for Output

We use iterators with **sequences** (also called **ranges**). These can be in containers, or they can be **input sequences** or **output sequences**. Figure 13.1 demonstrates

- input from the standard input (a sequence of data for program input), using an `istream_iterator`, and
- output to the standard output (a sequence of data for program output), using an `ostream_iterator`.

```
// fig13_01.cpp
// Demonstrating input and output with iterators.
#include <iostream>
#include <iterator> // ostream_iterator and istream_iterator

int main() {
   std::cout << "Enter two integers: ";

   // create istream_iterator for reading int values from cin
   std::istream_iterator <int> inputInt{std::cin};

   const int number1{*inputInt}; // read int from standard input
   ++inputInt; // move iterator to next input value
   const int number2{*inputInt}; // read int from standard input

   // create ostream_iterator for writing int values to cout
   std::ostream_iterator<int> outputInt{std::cout};

   std::cout << "The sum is: ";
   *outputInt = number1 + number2; // output result to cout
   std::cout << "\n";
}
```

```
Enter two integers: 25
The sum is: 37
```

Fig. 13.1 Demonstrating input and output with iterators.

The program inputs two integers from the user and displays their sum. As you'll see later in this chapter, `istream_iterator`s and `ostream_iterator`s can be used with the standard library algorithms to create powerful statements. For example, subsequent examples will use an `ostream_iterator` with the `copy` algorithm to copy a container's elements to the standard output stream with a single statement.

`istream_iterator`

Perf Line 10 creates an `istream_iterator` capable of **extracting** (inputting) `int` values from the standard input object `cin`. Line 12 **dereferences** iterator `inputInt` to read the first integer from `cin` and initializes `number1` with the value. The dereferencing operator `*` applied to `inputInt` gets the value from the stream. Line 13 positions `inputInt` to the next value in the input stream. Line 14 inputs the next `int` from `inputInt` and initializes `number2` with it. Either prefix or postfix increment can be used. We use the prefix form for performance reasons because it does not create a temporary object.

`ostream_iterator`

Line 17 creates an `ostream_iterator` capable of **inserting** (outputting) `int` values in the standard output object `cout`. Line 20 outputs an integer to `cout` by assigning to the dereferenced iterator (`*outputInt`) the sum of `number1` and `number2`.

13.3.2 Iterator Categories

The following table describes the iterator categories. Each provides a specific set of functionality. Throughout this chapter, we discuss which iterator category each container supports. In Chapter 14, you'll see that an algorithm's minimum iterator requirements determine which containers can be used with that algorithm.

Iterator category	Description
input	Used to read an element from a container. Can move only forward one element at a time from the container's beginning to its end. **Input iterators support only one-pass algorithms. The same input iterator cannot be used to pass through a sequence twice.**
output	Used to write an element to a container. Can move only forward one element at a time. **Output iterators support only one-pass algorithms. The same output iterator cannot be used to pass through a sequence twice.** For the subsequent iterator types in this table, if they refer to non-constant data, the iterators can be used to write to the container.
forward	Has the capabilities of **input and output iterators** and retains its position in the container. **Such iterators can pass through a sequence more than once for** multipass algorithms**.**
bidirectional	Has the capabilities of a **forward iterator** and adds the ability to move backward from the container's end toward its beginning. **Bidirectional iterators support multipass algorithms.**
random access	Has the capabilities of a **bidirectional iterator** and adds the ability to directly access any element of the container—that is, to jump forward or backward by an arbitrary number of elements. **These can also be compared with relational operators.**

Iterator category	Description
contiguous	20 A **random-access iterator** that requires elements to be stored in contiguous memory locations. This category was introduced in C++17 but formalized in C++20.

13.3.3 Container Support for Iterators

The iterator category that each container supports determines whether that container can be used with specific algorithms. **Containers that support random-access iterators can be used with all standard library algorithms**. Pointers into built-in arrays can be used as iterators. The following table shows the iterator category of each container. Sequence containers, associative containers, `string`s and built-in arrays are all traversable with iterators.

Container	Iterator type
Sequence containers	
`vector`	contiguous
`array`	contiguous
`deque`	random access
`list`	bidirectional
`forward_list`	forward
Ordered associative containers	
`set`	bidirectional
`multiset`	bidirectional
`map`	bidirectional
`multimap`	bidirectional
Unordered associative containers	
`unordered_set`	forward
`unordered_multiset`	forward

Container	Iterator type
`unordered_map`	forward
`unordered_multimap`	forward
Container adaptors	
`stack`	none
`queue`	none
`priority_queue`	none

13.3.4 Predefined Iterator Type Names

The following table shows the predefined iterator type names found in the standard library container class definitions. Not every iterator type name is defined for every container. **The `const` iterators are for traversing containers that should not be modified**. **Reverse iterators** traverse containers in the reverse direction.

Predefined iterator type name	Direction of ++	Capability
`iterator`	forward	read/write
`const_iterator`	forward	read
`reverse_iterator`	backward	read/write
`const_reverse_iterator`	backward	read

Operations performed on a `const_iterator` return references to `const` to prevent modifying container elements. Using `const_iterator`s where appropriate is another example of the principle of least privilege.

13.3.5 Iterator Operators

The following table shows operators for each iterator type. In addition to the operators shown for all iterators, iterators must provide **default constructors** (forward iterators and higher), **copy constructors** and **copy assignment operators**. A **forward iterator** supports `++` and all **input and output iterator** capabilities. A **bidirectional** iterator supports `--` and all the capabilities of **forward** iterators. **Random-access iterators** and **contiguous iterators** support all of the operations shown in the table. For input iterators and output iterators, it's not possible to save the iterator, then use the saved value later.

Iterator operation	Description
All iterators	
`++p`	Preincrement an iterator.
`p++`	Postincrement an iterator.
`p = p1`	Assign one iterator to another.
Input iterators	
`*p`	Dereference an iterator as a `const` *lvalue.*
`p->m`	Use the iterator to read the element m.
`p == p1`	Compare iterators for equality.
`p ! = p1`	Compare iterators for inequality.
Output iterators	
`*p`	Dereference an iterator as an *lvalue.*
`p = p1`	Assign one iterator to another.
Forward iterators	Forward iterators provide all the functionality of both input iterators and output iterators.
Bidirectional iterators	
`--p`	Predecrement an iterator.
`p--`	Postdecrement an iterator.
Random-access iterators	
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.

Iterator operation	Description
`p + i` *or* `i + p`	Expression value is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Expression value is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Expression value is an integer representing the distance (that is, number of elements) between two elements in the same container.
`p[i]`	Return a reference to the element offset from `p` by `i` positions
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (i.e., iterator p is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (i.e., iterator `p` is before iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (i.e., iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (i.e., iterator `p` is after iterator `p1` or at the same location as iterator `p1` in the container); otherwise, return `false`.

13.4 A Brief Introduction to Algorithms

20 The standard library provides scores of **algorithms** you can use to manipulate a wide variety of containers. Inserting, deleting, searching, sorting and others are appropriate for some or all of the sequence and associative containers. **The algorithms operate on container elements only indirectly through iterators.** Many algorithms operate on sequences of elements defined by iterators pointing to the **first element** of the sequence and to **one element past the last element**—known as **half-open ranges**. Many now support C++20 ranges. It's also possible to create your own new algorithms that operate in a similar fashion so they can be used with the standard library containers and iterators. In this chapter, we'll use the `copy` algorithm in many examples to copy a container's contents to the standard output. We discuss many standard library algorithms in Chapter 14.

13.5 Sequence Containers

The C++ standard library provides five **sequence containers**—`array`, `vector`, `deque`, `list` and `forward_list`.
The `array`, `vector` and `deque` containers are typically based on builtin arrays.
The `list` and `forward_list` containers implement linked-list data structures. We've already discussed

and used `array` extensively in Chapter 6, so we do not cover it again here. We've also already introduced `vector` in Chapter 6—and we discuss it in more detail here.

Perf Performance and Choosing the Appropriate Container

CG Section 13.2.2 presented the operations common to most standard library containers. Beyond these operations, each container typically provides various other capabilities. Many are common to several containers, but they're not always equally efficient. The C++ Core Guidelines say **`vector` is satisfactory for most applications.**[10] However, you should profile your application to ensure the container you choose is the correct one for your use-case and performance requirements.

10. C++ Core Guidelines, "SL.con.2: Prefer Using STL `vector` By Default Unless You Have a Reason to Use a Different Container." Accessed January 22, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rsl-vector.

Perf **Insertion at the back of a `vector` is efficient.** The `vector` grows, if necessary, to accommodate the new item. It's expensive to insert (or delete) an element in the middle of a `vector`. Every item after the insertion (or deletion) point must be moved because `vector` elements occupy contiguous cells in memory.

Perf For applications that require frequent insertions and deletions at both ends of a container, consider a **`deque`** rather than a `vector`. Class `deque` is $O(1)$ for insertions and deletions at the front,[11] making it more efficient than `vector`, which is $O(n)$ for those operations.[12] If you're not familiar with the **Big *O* notation** used here, see Section 13.12 for a friendly introduction.

11. "`std::deque`." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/deque.
12. "`std::vector`." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/vector.

Perf Applications with frequent insertions and deletions in the middle and/or at the extremes of a container sometimes use a **`list`** due to its efficient implementation of insertion and deletion anywhere in the data structure.

13.6 `vector` Sequence Container

Perf Perf The **`vector`** container (introduced in Section 6.15) is a dynamic data structure with contiguous memory locations. It provides rapid indexed access with the overloaded subscript operator `[]` like a built-in array or an `array` object. **Choose the `vector` container for the best random-access performance in a container that can grow.** When a `vector`'s capacity is exceeded, it

- **allocates** a larger built-in array,
- **copies** or **moves** (depending on what the element type supports) the original elements into the new built-in array and
- **deallocates** the old built-in array.

13.6.1 Using `vector`s and Iterators

Figure 13.2 illustrates several `vector` member functions, many of which are available in every sequence container and associative container. As we add items to a `vector<int>` in this example, we call our `showResult` function (lines 9–12) to display the new value as well as the `vector`'s

- size—the number of elements it currently contains and
- capacity—the number of elements it can store before it needs to **dynamically resize itself** to accommodate more elements.

```
1  // fig13_02.cpp
2  // Standard library vector class template.
3  #include <fmt/format.h> // C++20: This will be #include <format>
4  #include <iostream>
5  #include <ranges>
6  #include <vector> // vector class-template definition
7
8  // display value appended to vector and updated vector size and capacity
9  void showResult(int value, size_t size, size_t capacity) {
10    std::cout << fmt::format("appended: {}; size: {}; capacity: {}\n",
11                  value, size, capacity);
12 }
13
```

Fig. 13.2 Standard library `vector` class template.

Creating a `vector` and Displaying Its Initial Size and Capacity

Line 15 defines the `vector<int>` object `integers`. `vector`'s default constructor creates an empty `vector` with its size and capacity set to 0. So, the `vector` will have to allocate memory when elements are added to it. Lines 17–18 display the `size` and `capacity`. Function `size` returns the number of elements currently stored in the container.[13] Function **`capacity`** returns the `vector`'s current capacity.

13. `forward_list` does not have the `size` member function.

```
14 int main() {
15    std::vector<int> integers{}; // create vector of ints
16
17    std::cout << "Size of integers: " << integers.size()
18       << "\nCapacity of integers: " << integers.capacity() << "\n\n";
19
```

```
Size of integers: 0
Capacity of integers: 0
```

`vector` Member Function `push_back`

Lines 21–24 call **`push_back`** to append one element at a time to the `vector`. Each loop iteration calls `showResult` to display the added item and the `vector`'s new `size` and `capacity`. Function `push_back` is available in sequence containers other than `array` and `forward_list`. Sequence containers other than `array` and `vector` also provide a **`push_front`** function. If the `vector`'s **`size` equals its `capacity`** when you add a new element, the `vector` increases its capacity to accommodate more elements.

```
20    // append 1-10 to integers and display updated size and capacity
21    for (int i : std::views::iota(1, 11)) {
22       integers.push_back(i); // push_back is in vector, deque and list
23       showResult(i, integers.size(), integers.capacity());
24    }
25
```

```
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 3
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 6
appended: 6; size: 6; capacity: 6
```

```
appended: 7; size: 7; capacity: 9
appended: 8; size: 8; capacity: 9
appended: 9; size: 9; capacity: 9
appended: 10; size: 10; capacity: 13
```

Updated `size` and `capacity` After Modifying a `vector`

Perf The C++ standard does not specify how a `vector` grows to accommodate more elements—a time-consuming operation. **Some implementations double the `vector`'s capacity**. Others increase the capacity by 1.5 times, as shown in the output from **Visual C++**. This becomes apparent starting when the size and capacity are both 4:

- When we appended 5, the `vector` increased the capacity from 4 to 6, and the size became 5, leaving room for one more element.
- When we appended 6, the capacity remained at 6, and the size became 6.
- When we appended 7, the `vector` increased the capacity from 6 to 9, and the size became 7, leaving room for two more elements.
- When we appended 8, the capacity remained at 9, and the size became 8.
- When we appended 9, the capacity remained at 9, and the size became 9.
- When we appended 10, the `vector` increased the capacity from 9 to 13 (1.5 times 9 rounded down to the nearest integer), and the size became 10, leaving room for three more elements before the `vector` will need to allocate more space.

`vector` Growth in `g++`

C++ library implementers use various schemes to minimize `vector` resizing overhead, so this program's output may vary based on your compiler's `vector` growth implementation. **GNU `g++`**, for example, **doubles a `vector`'s capacity** when more room is needed, producing the following output:

```
appended: 1; size: 1; capacity: 1
appended: 2; size: 2; capacity: 2
appended: 3; size: 3; capacity: 4
appended: 4; size: 4; capacity: 4
appended: 5; size: 5; capacity: 8
appended: 6; size: 6; capacity: 8
appended: 7; size: 7; capacity: 8
appended: 8; size: 8; capacity: 8
appended: 9; size: 9; capacity: 16
appended: 10; size: 10; capacity: 16
```

Other `vector` Sizing Considerations

Perf Some programmers allocate a large initial capacity. If a `vector` stores a small number of elements, such capacity may waste space. However, it can greatly improve performance if a program adds many elements to a `vector` and does not have to reallocate memory to accommodate those elements. This is a classic space–time trade-off. Library implementors must balance the amount of memory used against the time required to perform various `vector` operations.

Perf It can be wasteful to double a `vector`'s size when more space is needed. For example, in a `vector` implementation that doubles the allocated memory when space is needed, a full `vector` of 1,000,000 elements resizes to accommodate 2,000,000 elements even when only one new element is

added. This leaves 999,999 unused elements. You can use member functions `reserve` to control space usage better.

Outputting `vector` Contents with Iterators

Lines 28–31 output the `vector`'s contents. Line 28 initializes the control variable `constIterator` using `vector` member function **`cbegin`**, which returns a **`const_iterator`** to the `vector`'s first element. We use **`auto`** to infer the control variable's type

```
vector<int>::const_iterator
```

which simplifies the code and reduces errors when working with more complex types.

```
26     std::cout << "\nOutput integers using iterators: ";
27
28     for (auto constIterator{integers.cbegin()};
29        constIterator != integers.cend(); ++constIterator) {
30        std::cout << *constIterator << ' ';
31     }
32
Output integers using iterators: 1 2 3 4 5 6 7 8 9 10
```

Err The loop continues as long as `constIterator` has not reached the `vector`'s end. This is determined by comparing `constIterator` to the result of calling the `vector`'s **`cend`** function. When `constIterator` equals this value, the loop terminates. **Attempting to dereference an iterator positioned outside its container is a logic error. The iterator returned by `end` or `cend` should never be dereferenced or incremented.**

Line 30 dereferences `constIterator` to get the current element's value. Remember that the iterator acts like a pointer to an element, and operator `*` is overloaded to return a reference to the element. The expression `++constIterator` (line 29) positions the iterator to the `vector`'s next element. You could replace this loop with the following more straightforward range-based `for` statement:

```
for (auto const& item : integers) {
   std::cout << item << ' ';
}
```

The range-based `for` uses iterators "under the hood."

Displaying the `vector`'s Contents in Reverse with `const_reverse_iterators`

Lines 36–39 iterate through the `vector` in reverse. The `vector` member functions **`crbegin`** and **`crend`** each return **`const_reverse_iterators`** representing the starting and ending points when iterating through a container in reverse. Most sequence containers and the ordered associative containers support reverse iteration. Class `vector` also provides member functions **`rbegin`** and **`rend`** to obtain **non-`const` `reverse_iterators`**.

```
33     std::cout << "\nOutput integers in reverse using iterators: ";
34
```

```
35      // display vector in reverse order using const_reverse_iterator
36      for (auto reverseIterator{integers.crbegin()};
37         reverseIterator != integers.crend(); ++reverseIterator) {
38         std::cout << *reverseIterator << ' ';
39      }
40
41      std::cout << "\n";
42   }
```

```
Output integers in reverse using iterators: 10 9 8 7 6 5 4 3 2 1
```

11 C++11: `shrink_to_fit`

As of C++11, you can call member function **`shrink_to_fit`** to request that a `vector` or `deque` return unneeded memory to the system, reducing its capacity to the container's current number of elements. According to the C++ standard, implementations can ignore this request so that they can perform implementation-specific optimizations.

*13.6.2 **`vector`** Element-Manipulation Functions*

Figure 13.3 illustrates functions for retrieving and manipulating `vector` elements. Line 12 initializes a `vector<int>` with a braced initializer. In this case, we declared the `vector`'s type as `std::vector`. The compiler uses class template argument deduction (CTAD) to infer the element type from the initializer list's `int` values. Line 13 uses a `vector` constructor that takes two iterators as arguments to initialize `integers` with a copy of the elements in the range `values.cbegin()` up to, but not including, `values.cend()`.

```
1  // fig13_03.cpp
2  // Testing standard library vector class template
3  // element-manipulation functions.
4  #include <algorithm> // copy algorithm
5  #include <fmt/format.h> // C++20: This will be #include <format>
6  #include <iostream>
7  #include <ranges>
8  #include <iterator> // ostream_iterator iterator
9  #include <vector>
10
11 int main() {
12    std::vector values{1, 2, 3, 4, 5}; // class template argument deduction
13    std::vector<int> integers{values.cbegin(), values.cend()};
14    std::ostream_iterator<int> output{std::cout, " "};
15
```

Fig. 13.3 `vector` container element-manipulation functions.

`ostream_iterator`

Line 14 defines an `ostream_iterator` called `output` that can be used to output integers separated by single spaces via `cout`. An `ostream_iterator<int>` outputs only values of type `int`. The constructor's first argument specifies the output stream, and the second argument is a string specifying the separator for the values output—in this case, a space character. We use the `ostream_iterator` (header `<iterator>`) to output the `vector` contents in this example.

`copy` Algorithm

Line 17 uses standard library algorithm **`copy`** (header **`<algorithm>`**) to output the contents of `integers` to the standard output. This version of the algorithm takes three arguments. The first two are iterators that specify the elements to copy from `vector integers`—from `integers.cbegin()` up to, but not including, **`integers.cend()`**. These must satisfy **input iterator** requirements, such as **`const_iterator`**s, enabling values to be read from a container. They must also refer to the same container such that applying `++` to the first iterator repeatedly causes it to reach the second iterator argument eventually. The third argument specifies where to copy the elements. This must be an **output iterator** through which a value can be stored or output. The output iterator is an **`ostream_iterator`** attached to `cout`, so line 17 copies the elements to the standard output.

```
16     std::cout << "integers contains: ";
17     std::copy(integers.cbegin(), integers.cend(), output);
18
integers contains: 1 2 3 4 5
```

Common Ranges

Before C++20, a "range" of container elements was described by iterators specifying the starting position and the one-past-the-end position, typically via calls to a container's `begin` and `end` (or similar) member functions. As of C++20, the C++ standard refers to such ranges as **common ranges**, so they're not confused with the new C++20 ranges capabilities. From this point forward, we'll use the term "common range" when discussing ranges determined by two iterators, and we'll use the term "range" when discussing C++20 ranges.

`vector` Member Functions `front` and `back`

Err Lines 19–20 get the `vector`'s first and last elements via member functions **`front`** and **`back`**, which are available in most sequence containers. Notice the difference between functions `front` and `begin`. Function `front` returns a reference to the `vector`'s first element, while function `begin` returns an iterator pointing to the `vector`'s first element. Similarly, function **`back`** returns a reference to the `vector`'s last element, whereas **`end`** returns an iterator pointing to the location after the last element. The results of **`front`** and **`back`** are undefined when called on an empty `vector`.

```
19     std::cout << fmt::format("\nfront: {}\nback: {}\n\n",
20        integers.front(), integers.back());
21
front: 1
back: 5
```

Accessing `vector` Elements

Lines 22–23 illustrate two ways to access `vector` (or `deque`) elements. Line 22 uses the `[]` operator, which returns a reference to the value at the specified location or a reference to that `const` value, depending on whether the container is `const`. Function `at` (line 23) performs the same operation but with **bounds checking**. The `at` member function first checks its argument and determines whether it's in the `vector`'s bounds. If not, function `at` throws an **`out_of_range`** exception, as demonstrated in Section 6.15.

```
22      integers[0] = 7; // set first element to 7
23      integers.at(2) = 10; // set element at position 2 to 10
24
```

`vector` Member Function `insert`

Line 26 uses one of the several overloaded **`insert` member functions** provided by each sequence container (except **`array`**, which has a fixed size, and **`forward_list`**, which has the function `insert_after` instead). Line 26 inserts the value 22 before the element at the location specified by the iterator in the first argument. Here, the iterator points to the second element, so 22 becomes the second element, and the original second element is now the third. Other versions of `insert` allow

- inserting multiple copies of the same value starting at a given position or
- inserting a range of values from another container, starting at a given position.

The version of **`insert`** in line 26 returns an iterator pointing to the inserted item.

```
25      // insert 22 as second element
26      integers.insert(integers.cbegin() + 1, 22);
27
28      std::cout << "Contents of vector integers after changes: ";
29      std::ranges::copy(integers, output);
30
```

```
Contents of vector integers after changes: 7 22 2 10 4 5
```

20 C++20 Ranges Algorithm `copy`

Line 29 uses the C++20 **`copy` algorithm** from the **`std::ranges` namespace** to copy the elements of `integers` to the standard output. **This version of `copy` receives only the range to copy and the output iterator representing where to copy the range's elements.** The first argument is an object representing a **range of elements** with **input iterators** representing its beginning and end—in this case, a `vector`. Most pre-C++20 algorithms in the **`<algorithm>`** header now have C++20 ranges versions in the **`std::ranges`** namespace. The **`std::ranges`** algorithms typically are overloaded with a version that takes a **range object** and a version that takes an **iterator** and a **sentinel**. In C++20 ranges, a **sentinel** is an object that represents when the end of the container has been reached. Chapter 14 demonstrates many C++20 ranges algorithms.

`vector` Member Function `erase`

Lines 31 and 36 use two **`erase`** member functions that are available in most sequence and associative containers—except **`array`**, which has a fixed size, and **`forward_list`**, which has the function **`erase_after`** instead. Line 31 erases the element at the location specified by its iterator argument—in this case, the first element. Line 36 erases elements in the common range specified by the two iterator arguments—in this case, all the elements. Line 38 uses member function **`empty`** (defined for all containers and adaptors) to confirm that the `vector` is empty.

```
31      integers.erase(integers.cbegin()); // erase first element
32      std::cout << "\n\nintegers after erasing first element: ";
33      std::ranges::copy(integers, output);
34
35      // erase remaining elements
36      integers.erase(integers.cbegin(), integers.cend());
```

```
37      std::cout << fmt::format("\nErased all elements: integers {} empty\n",
38         integers.empty() ? "is" : "is not");
39
```

```
integers after erasing first element: 22 2 10 4 5
Erased all elements: integers is empty
```

Err Usually, `erase` destroys the objects it erases. However, **erasing an element that is a pointer to a dynamically allocated object does not `delete` the object, potentially causing a memory leak.** Again, this is why you should not manage dynamically allocated memory with raw pointers.
A **`unique_ptr`** (Section 11.5) element would release the dynamically allocated memory. If the element is a **`shared_ptr`** (online Chapter 20), the reference count to the dynamically allocated object would be decremented, and the memory would be deleted only if the reference count reached `0`.

`vector` Member Function `insert` with Three Arguments (Range `insert`)

Line 41 demonstrates the version of function **`insert`** that uses its second and third arguments to specify a common range of elements to insert into the `vector` (in this case, from `values`). Remember that the ending location specifies the position in the sequence **after** the last element to insert; copying occurs up to, but not including, this location. This version of member function **`insert`** returns an iterator pointing to the first item that was inserted. If nothing was inserted, the function returns its first argument.

```
40      // insert elements from the vector values
41      integers.insert(integers.cbegin(), values.cbegin(), values.cend());
42      std::cout << "\nContents of vector integers before clear: ";
43      std::ranges::copy(integers, output);
44
```

```
Contents of vector integers before clear: 1 2 3 4 5
```

`vector` Member Function `clear`

Finally, line 46 uses member function **`clear`** to empty the `vector`. All sequence containers and associative containers except `array`, which is fixed in size, provide member function `clear`. **This does not reduce the `vector`'s capacity.**

```
45      // empty integers; clear empties a collection
46      integers.clear();
47      std::cout << fmt::format("\nAfter clear, integers {} empty\n",
48                      integers.empty() ? "is" : "is not");
49   }
```

```
After clear, integers is empty
```

13.7 `list` Sequence Container

Perf The **`list` sequence container** (from header **`<list>`**) allows insertion and deletion operations at any location in the container. The `list` container is implemented as a **doubly linked list**[14]—every node in the `list` contains a pointer to the previous node in the **`list`** and to the next node in the `list`. This enables `list`s to support **bidirectional iterators** that allow the container to be traversed both forward and backward. Any algorithm that requires **input**, **output**, **forward** or **bidirectional**

iterators can operate on a `list`. Many `list` member functions manipulate the elements of the container as an ordered set of elements. If most of the insertions and deletions occur at the ends of the container, consider using **`deque`** (Section 13.8) or **`vector`**. Recall that `deque`'s implementation of inserting at the front is more efficient than `vector`'s.
14. "`std::list`." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/list.

`forward_list` Container

The **`forward_list`** sequence container (**header `<forward_list>`**; added in C++11) is implemented as a **singly linked list**—every node contains a pointer to the next node in the `forward_list`. This enables a `forward_list` to support **forward iterators** that allow the container to be traversed in the forward direction. Any algorithm that requires **input**, **output** or **forward iterators** can operate on a `forward_list`.

`list` Member Functions

Perf In addition to the common sequence container member functions discussed in Section 13.2, **`list`** provides member functions **`splice`**, **`push_front`**, **`pop_front`**, **`emplace_front`**, **`emplace_back`**, **`remove`**, **`remove_if`**, **`unique`**, **`merge`**, **`reverse`** and **`sort`**. Several of these are `list`-optimized implementations of standard library algorithms we show in Chapter 14. Both **`push_front`** and **`pop_front`** are also supported by **`forward_list`** and **`deque`**. Figure 13.4 demonstrates several `list` features. Many of the functions presented in Figs. 13.2–13.3 also can be used with `list`s. We focus on the new features in this example's discussion.

```
1  // fig13_04.cpp
2  // Standard library list class template.
3  #include <algorithm> // copy algorithm
4  #include <iostream>
5  #include <iterator> // ostream_iterator
6  #include <list> // list class-template definition
7  #include <vector>
8
9  // printList function template definition; uses
10 // ostream_iterator and copy algorithm to output list elements
11 template <typename T>
12 void printList(const std::list<T>& items) {
13    if (items.empty()) { // list is empty
14       std::cout << "List is empty";
15    }
16    else {
17       std::ostream_iterator<T> output{std::cout, " "};
18       std::ranges::copy(items, output);
19    }
20 }
21
```

Fig. 13.4 Standard library `list` class template.

Function template `printList` (lines 11–20) checks whether its `list` argument is `empty` (line 13) and, if so, displays an appropriate message. Otherwise, `printList` uses an **`ostream_iterator`** and the **`std::ranges::copy`** algorithm to copy the `list`'s elements to the standard output, as shown in Fig. 13.3.

Creating a `list` Object

Line 23 creates a `list` object capable of storing `int`s. Lines 26–27 use its member function `push_front` to insert integers at the beginning of `values`. This member function is specific to classes `forward_list`, `list` and `deque`. Lines 28–29 use `push_back` to append integers to `values`. **Function `push_back` is common to all sequence containers**, except `array` and `forward_list`. Lines 31–32 show the current contents of `values`.

```
22  int main() {
23     std::list<int> values{}; // create list of ints
24
25     // insert items in values
26     values.push_front(1);
27     values.push_front(2);
28     values.push_back(4);
29     values.push_back(3);
30
31     std::cout << "values contains: ";
32     printList(values);
33
values contains: 2 1 4 3
```

`list` Member Function `sort`

Line 34 uses `list` member function `sort` to arrange the elements in ascending order. A second version of `sort` allows you to supply a **binary predicate function** that takes two arguments (values in the `list`), performs a comparison and returns a `bool` value indicating whether the first argument should come before the second in the sorted contents. This function determines the order in which the elements are sorted. This version is useful for sorting a `list` in descending order or customizing how elements are compared to determine sort order.

```
34     values.sort(); // sort values
35     std::cout << "\nvalues after sorting contains: ";
36     printList(values);
37
values after sorting contains: 1 2 3 4
```

`list` Member Function `splice`

Line 46 uses `list` member function `splice` to remove the elements in `otherValues` and insert them into `values` before the iterator position specified as the first argument. Function `splice` with three arguments allows one element to be removed from the container specified as the second argument from the location specified by the iterator in the third argument. Function `splice` with four arguments uses the last two arguments to specify a common range to remove from the `list` in the second argument and insert at the location specified in the first argument. A `forward_list` provides a similar member function named `splice_after`.

```
38     // insert elements of ints into otherValues
39     std::vector ints{2, 6, 4, 8};
40     std::list<int> otherValues{}; // create list of ints
```

```
41     otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
42     std::cout << "\nAfter insert, otherValues contains: ";
43     printList(otherValues);
44
45     // remove otherValues elements and insert at end of values
46     values.splice(values.cend(), otherValues);
47     std::cout << "\nAfter splice, values contains: ";
48     printList(values);
49
```

```
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
```

`list` Member Function `merge`

After inserting more elements in `otherValues` and sorting both `values` and `otherValues`, line 61 uses `list` member function **`merge`** to remove all elements of `otherValues` and **insert them in sorted order** into `values`. Both `list`s must be sorted in the same order before this operation is performed. A second version of `merge` enables you to supply a **binary predicate function** that takes two arguments (values in the `list`) and returns a `bool` value. The predicate function specifies the sorting order used by `merge`, returning `true` if the predicate's first argument should be placed before the second.

```
50     values.sort(); // sort values
51     std::cout << "\nAfter sort, values contains: ";
52     printList(values);
53
54     // insert elements of ints into otherValues
55     otherValues.insert(otherValues.cbegin(), ints.cbegin(), ints.cend());
56     otherValues.sort(); // sort the list
57     std::cout << "\nAfter insert and sort, otherValues contains: ";
58     printList(otherValues);
59
60     // remove otherValues elements and insert into values in sorted order
61     values.merge(otherValues);
62     std::cout << "\nAfter merge:\n values contains: ";
63     printList(values);
64     std::cout << "\n otherValues contains: ";
65     printList(otherValues);
66
```

```
After sort, values contains: 1 2 2 3 4 4 6 8
After insert and sort, otherValues contains: 2 4 6 8
After merge:
   values contains: 1 2 2 2 3 4 4 4 6 6 8 8
   otherValues contains: List is empty
```

`list` Member Function `pop_front` and `pop_back`

Line 67 uses `list` member function **`pop_front`** to remove the first element. Line 68 uses function **`pop_back`** to remove the last element. This function is available for **sequence containers**, except **`array`** and **`forward_list`**.

```
67     values.pop_front(); // remove element from front
68     values.pop_back(); // remove element from back
```

```
69      std::cout << "\nAfter pop_front and pop_back:\n values contains: ";
70      printList(values);
71
```

```
After pop_front and pop_back:
   values contains: 2 2 2 3 4 4 4 6 6 8
```

`list` Member Function `unique`

Line 72 uses `list` function **`unique`** to **remove duplicate adjacent elements**. If the `list` is sorted, **all duplicates are eliminated**. A second version of `unique` enables you to supply a **predicate function** that takes two arguments (values in the **`list`**) and returns a `bool` value specifying whether two elements are equal.

```
72      values.unique(); // remove duplicate elements
73      std::cout << "\nAfter unique, values contains: ";
74      printList(values);
75
```

```
After unique, values contains: 2 3 4 6 8
```

`list` Member Function `swap`

Line 76 uses `list` member function **`swap`** to exchange the contents of `values` with the contents of `otherValues`. This function is available to all **sequence containers** and **associative containers**.

```
76      values.swap(otherValues); // swap elements of values and otherValues
77      std::cout << "\nAfter swap:\n values contains: ";
78      printList(values);
79      std::cout << "\n otherValues contains: ";
80      printList(otherValues);
81
```

```
After swap:
   values contains: List is empty
   otherValues contains: 2 3 4 6 8
```

`list` Member Functions `assign` and `remove`

Line 83 uses `list` member function **`assign`** (available in all **sequence containers**) to replace `values`' contents with a common range of elements from `otherValues`. A second version of `assign` replaces the `list`'s contents with the specified number of copies of the value specified in its second argument. Line 92 uses `list` member function **`remove`** to **delete all copies** of the value `4` from the `list`.

```
82      // replace contents of values with elements of otherValues
83      values.assign(otherValues.cbegin(), otherValues.cend());
84      std::cout << "\nAfter assign, values contains: ";
85      printList(values);
86
87      // remove otherValues elements and insert into values in sorted order
88      values.merge(otherValues);
89      std::cout << "\nAfter merge, values contains: ";
90      printList(values);
```

```
91
92     values.remove(4); // remove all 4s
93     std::cout << "\nAfter remove(4), values contains: ";
94     printList(values);
95     std::cout << "\n";
96  }
```

```
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove(4), values contains: 2 2 3 3 6 6 8 8
```

13.8 deque Sequence Container

Perf Perf Class **deque** (header **<deque>**)—short for "double-ended queue"—provides many benefits of vectors and lists in one container. It's implemented to provide efficient indexed access (using subscripting) for reading and modifying elements, like a vector or array. It also provides **efficient insertion and deletion at its front and back**, like a list.
Class deque supports **random-access iterators**, so deques can be used with all standard library algorithms. A common use of a deque is to maintain a first-in, first-out queue of elements. In fact, a deque is the default container for the queue adaptor[15] (Section 13.10.2).

15. "std::queue." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/queue.

Additional storage for a deque can be allocated at either end in blocks of memory that are typically maintained as a **built-in array of pointers to those blocks**.[16] Due to a deque's **noncontiguous memory layout**, its iterators must be more "intelligent" than those for vectors, arrays or built-in arrays.
A deque provides the same operations as vector, but like list, adds member functions **push_front** and **pop_front** for efficient insertion and deletion at the beginning of the deque.

16. This is an implementation-specific detail, not a requirement of the C++ standard.

Figure 13.5 demonstrates several deque features. Many functions presented in Figs. 13.2–Fig. 13.4 also can be used with deque.

```
1  // fig13_05.cpp
2  // Standard library deque class template.
3  #include <algorithm> // copy algorithm
4  #include <deque> // deque class-template definition
5  #include <iostream>
6  #include <iterator> // ostream_iterator
7
8  int main() {
9     std::deque<double> values; // create deque of doubles
10    std::ostream_iterator<double> output{std::cout, " "};
11
12    // insert elements in values
13    values.push_front(2.2);
14    values.push_front(3.5);
15    values.push_back(1.1);
16
17    std::cout << "values contains: ";
18
19    // use subscript operator to obtain elements of values
20    for (size_t i{0}; i < values.size(); ++i) {
```

```
21         std::cout << values[i] << ' ';
22      }
23
24      values.pop_front(); // remove first element
25      std::cout << "\nAfter pop_front, values contains: ";
26      std::ranges::copy(values, output);
27
28      // use subscript operator to modify element at location 1
29      values[1] = 5.4;
30      std::cout << "\nAfter values[1] = 5.4, values contains: ";
31      std::ranges::copy(values, output);
32      std::cout << "\n";
33   }
```

```
values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[1] = 5.4, values contains: 2.2 5.4
```

Fig. 13.5 Standard library `deque` class template.

Line 9 instantiates a `deque` that can store `double` values, then lines 13–15 use member functions **`push_front`** and **`push_back`** to insert elements at its beginning and end.

Lines 20–22 use the subscript operator to retrieve each element's value for output. The loop condition uses member function `size` to ensure that we do not attempt to access an element outside the `deque`'s bounds. We use a counter-controlled `for` loop here only to demonstrate the `[]` operator. Generally, you should use the range-based `for` to process all the elements of a container.

Line 24 uses member function **`pop_front`** to demonstrate removing the `deque`'s first element. Line 29 uses the subscript operator to obtain an *lvalue*, which we use to assign a new value to element `1` of the `deque`.

13.9 Associative Containers

The **associative containers** provide **direct access** to store and retrieve elements via **keys**. The four **ordered associative containers** are **`multiset`**, **`set`**, **`multimap`** and **`map`**. Each of these maintains its keys in sorted order. There are also four corresponding **unordered associative containers**—**`unordered_multiset`**, **`unordered_set`**, **`unordered_multimap`** and **`unordered_map`**—that offer most of the same capabilities as their ordered counterparts. The primary difference between ordered and unordered associative containers is that unordered ones do not maintain their keys in sorted order. If your keys are not comparable, you must use the unordered containers. This section focuses on the **ordered associative containers**.

Perf For cases in which it's not necessary to maintain keys in sorted order, the **unordered associative containers** offer better search performance via **hashing**—***O*(1)** with a worst-case of ***O*(*n*)** vs. ***O*(log *n*)** for the ordered associative containers.[17] However, this requires the keys to be hashable. For hashable-type requirements, see the documentation for `std::hash`.[18] For an introduction to hashing, see Section 13.13.

17. "Containers Library." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container.
18. "std::hash." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/utility/hash.

17 Iterating through an **ordered associative container** traverses it in the sort order for that container. Classes **`multiset`** and **`set`** provide operations for manipulating sets of values where the values themselves are the keys. **The primary difference between a `multiset` and a `set` is that a `multiset` allows duplicate keys and a `set` does not.** Classes **`multimap`** and **`map`** provide operations for manipulating values associated with keys (these values are sometimes referred to as **mapped**

values). **The primary difference between a `multimap` and a `map` is that a `multimap` allows duplicate keys and a `map` allows only unique keys.** In addition to the common container member functions, **ordered associative containers** support several member functions specific to associative containers. You can combine the contents of associative containers of the same type using the `merge` function (added in C++17). Examples of the **ordered associative containers** and their common member functions are presented in the next several subsections.

13.9.1 `multiset` Associative Container

The **`multiset` ordered associative container** (from header **`<set>`**) provides fast storage and retrieval of keys and allows duplicate keys. The elements' ordering is determined by a **comparator function object**. A **function object** is an instance of a class that has an overloaded parentheses operator, allowing the object to be "called" like a function. For example, in an integer **`multiset`**, elements can be sorted in ascending order by ordering the keys with **comparator function object `less<int>`**, which knows how to compare two `int` values to determine whether the first is less than the second. This enables an integer **`multiset`** to order its elements in ascending order. Section 14.5 discusses function objects in detail. Here, we'll simply show how to use **`less<int>`** when declaring ordered associative containers.

The data type of the keys in all ordered associative containers must support comparison based on the **comparator function object**—keys sorted with **`less<T>`** must support comparison with **`operator<`**. If the keys used in the ordered associative containers are of user-defined data types, those types must supply comparison operators. A **`multiset`** supports **bidirectional iterators**. If the order of the keys is not important, consider **`unordered_multiset`** (header `<unordered_set>`), but keep in mind that it supports only **forward iterators**.

Creating a `multiset`

Figure 13.6 demonstrates the **`multiset` ordered associative container** with `int` keys that are sorted in ascending order. Containers **`multiset`** and **`set`** (Section 13.9.2) provide the same basic functionality. Line 12 creates the `multiset`, using the function object `std::less<int>` to specify the keys' sort order. The compiler knows this `multiset` contains `int` values, so you can specify `std::less<int>` simply as `std::less<>`—the compiler will infer that `std::less` compares `int` values. Also, `std::less<>` is the default for a **`multiset`**, so line 12 can be simplified as

```
std::multiset<int> ints{}; // multiset of int values
```

```
1  // fig13_06.cpp
2  // Standard library multiset class template
3  #include <algorithm> // copy algorithm
4  #include <fmt/format.h> // C++20: This will be #include <format>
5  #include <iostream>
6  #include <iterator> // ostream_iterator
7  #include <ranges>
8  #include <set> // multiset class-template definition
9  #include <vector>
10
11 int main() {
12    std::multiset<int, std::less<int>> ints{}; // multiset of int values
```

Fig. 13.6 Standard library `multiset` class template.

`multiset` Member Function `count`

Line 13 uses function `count` (available to all associative containers) to count the number of occurrences of the value `15` currently in the `multiset`.

```
13     std::cout << fmt::format("15s in ints: {}\n", ints.count(15));
14
```

```
15s in ints: 0
```

`multiset` Member Function `insert`

Lines 16–17 use one of the several overloaded versions of member function `insert` to add the value `15` to the `multiset` twice. A second version of `insert` takes an iterator and a value as arguments and begins the search for the insertion point from the iterator position specified. A third version of `insert` takes two iterators that specify a common range to add to the `multiset` from another container. For several other overloads, see

```
https://en.cppreference.com/w/cpp/container/multiset/insert
```

```
15     std::cout << "\nInserting two 15s into ints\n";
16     ints.insert(15); // insert 15 in ints
17     ints.insert(15); // insert 15 in ints
18     std::cout << fmt::format("15s in ints: {}\n\n", ints.count(15));
19
```

```
Inserting two 15s into ints
15s in ints: 2
```

`multiset` Member Function `find`

Lines 21–28 use member function `find` (line 22), which is available to all **associative containers**, to search for the values `15` and 20 in the `multiset`. The range-based `for` loop iterates through each item in `{15, 20}`, which creates an `initializer_list`. Function `find` returns either an `iterator` or a `const_iterator`, depending on whether the `multiset` is `const`. The iterator points to the location at which the value is found. If the value is not found, `find` returns an `iterator` or a `const_iterator` equal to the value returned by the container's `end` member function. Usually, you'd use `find` if you need to use the iterator that points to the found element. Here, we could have used the `count` member function—if it returns `0`, the item is not in the `multiset`.

```
20     // search for 15 and 20 in ints; find returns an iterator
21     for (int i : {15, 20}) {
22        if (auto result{ints.find(i)}; result != ints.end()) {
23           std::cout << fmt::format("Found {} in ints\n", i);
24        }
25        else {
26           std::cout << fmt::format("Did not find {} in ints\n", i);
27        }
28     }
29
```

```
Found 15 in ints
Did not find 20 in ints
```

20 `multiset` Member Function `contains` (C++20)

Lines 31–38 use the new C++20 member function **`contains`** (line 32) to determine whether the values `15` and 20 are in the **`multiset`**. This function, which is available to all **associative containers**, returns a `bool` indicating whether the value is present in the container. The range-based `for` loop iterates through each item in `{15, 20}`.

```
30     // search for 15 and 20 in ints; contains returns a bool
31     for (int i : {15, 20}) {
32        if (ints.contains(i)) {
33           std::cout << fmt::format("Found {} in ints\n", i);
34        }
35        else {
36           std::cout << fmt::format("Did not find {} in ints\n", i);
37        }
38     }
39
```

```
Found 15 in ints
Did not find 20 in ints
```

Inserting Elements of Another Container into a `multiset`

Line 42 uses member function **`insert`** to insert a `vector`'s elements into the `multiset`, then line 44 copies the `multiset`'s elements to the standard output. The values display in ascending order because the `multiset` is an ordered container that maintains its elements in ascending order by default. In line 44, note the expression

```
std::ostream_iterator<int>{std::cout, " "}
```

This creates a temporary `ostream_iterator` and immediately passes it to `copy`. We chose this approach because the `ostream_iterator` is used only once in this example.

```
40     // insert elements of vector values into ints
41     const std::vector values{7, 22, 9, 1, 18, 30, 100, 22, 85, 13};
42     ints.insert(values.cbegin(), values.cend());
43     std::cout << "\nAfter insert, ints contains:\n";
44     std::ranges::copy(ints, std::ostream_iterator<int>{std::cout, " "});
45
```

```
After insert, ints contains:
1 7 9 13 15 15 18 22 22 30 85 100
```

`multiset` Member Functions `lower_bound` and `upper_bound`

Line 49 uses functions **`lower_bound`** and **`upper_bound`**, available in all **ordered associative containers**, to locate the earliest occurrence of the value `22` in the `multiset` and the element after the last occurrence of the value `22` in the `multiset`. Both functions return **`iterator`**s or **`const_iterator`**s pointing to the appropriate location, or they return the `end` iterator if the value is not in the `multiset`. Together, the lower bound and upper bound represent the common range of elements containing the value `22`.

```
46      // determine lower and upper bound of 22 in ints
47      std::cout << fmt::format(
48         "\n\nlower_bound(22): {}\nupper_bound(22): {}\n\n",
49         *ints.lower_bound(22), *ints.upper_bound(22));
50
```

```
lower_bound(22): 22
upper_bound(22): 30
```

`pair` Objects and `multiset` Member Function `equal_range`

The `multiset` function **`equal_range`** returns a **`pair`** containing the results of calling both **`lower_bound`** and **`upper_bound`**. A `pair` associates two values. Line 52 creates and initializes a `pair` object called `p`. We use `auto` to infer the variable's type from its initializer. The `pair` returned by **`equal_range`**, which will contain two **`iterator`**s or **`const_iterator`**s, depending on whether the `multiset` is `const`. A `pair` contains two `public` data members called **`first`** and **`second`**—their types depend on the `pair`'s initializers.

```
51      // use equal_range to determine lower and upper bound of 22 in ints
52      auto p{ints.equal_range(22)};
53      std::cout << fmt::format(
54         "lower_bound(22): {}\nupper_bound(22): {}\n",
55         *(p.first), *(p.second));
56   }
```

```
lower_bound(22): 22
upper_bound(22): 30
```

Line 52 uses `equal_range` to determine the `lower_bound` and `upper_bound` of `22` in the `multiset`. Line 55 uses `p.first` and `p.second` to access the `lower_bound` and `upper_bound`. We dereferenced the iterators to output the values at the locations returned from `equal_range`. Though we did not do so here, you should always ensure that the iterators returned by `lower_bound`, `upper_bound` and `equal_range` are not equal to the container's `end` iterator before dereferencing them.

C++14: Heterogeneous Lookup

14 Before C++14, when searching for a key in an associative container, the argument provided to a search function like `find` was required to have the container's key type. For example, if the key type were `string`, you could pass `find` a pointer-based string to locate in the container. In this case, the argument would be converted into a temporary object of the key type (`string`), then passed to `find`. In C++14 and higher, the argument to `find` (and other similar functions) can be of any type, provided that there are overloaded comparison operators that can compare values of the argument's type to values of the container's key type. If there are, no temporary objects will be created. This is known as **heterogeneous lookup**.

13.9.2 `set` Associative Container

The **`set` associative container** (from **header `<set>`**) is used for fast storage and retrieval of unique keys. The implementation of a `set` is identical to that of a `multiset`, except that a **`set` must have unique keys**. When a duplicate is inserted, it is ignored. This is the intended mathematical behavior of a set, so it's not considered an error. A `set` supports **bidirectional iterators**. If the order of the keys is not important, consider **`unordered_set`** (header `<unordered_set>`), but keep in mind that it supports only forward iterators, and its keys must be hashable.

Creating a `set`

Figure 13.7 demonstrates a `set` of `double`s. Line 10 creates the `set`, using class template argument deduction (CTAD) to infer the element type. Line 10 is equivalent to

```
std::set<double, std::less<double>> doubles{2.1, 4.2, 9.5, 2.1, 3.7};
```

```
 1  // fig13_07.cpp
 2  // Standard library set class template.
 3  #include <algorithm>
 4  #include <fmt/format.h> // C++20: This will be #include <format>
 5  #include <iostream>
 6  #include <iterator> // ostream_iterator
 7  #include <set>
 8
 9  int main() {
10     std::set doubles{2.1, 4.2, 9.5, 2.1, 3.7}; // CTAD
11
12     std::ostream_iterator<double> output{std::cout, " "};
13     std::cout << "doubles contains: ";
14     std::ranges::copy(doubles, output);
15
doubles contains: 2.1 3.7 4.2 9.5
```

Fig. 13.7 Standard library `set` class template.

20 The `set`'s **`initializer_list`** constructor inserts all the elements into the `set`. Line 14 uses the `std::ranges` algorithm `copy` to output the `set`'s contents. Notice that the value `2.1`, which appeared twice in the initializer list, appears only once in `doubles`. Again, **a `set` does not allow duplicates**.

Inserting a New Value into a `set`

Line 19 defines and initializes a `pair` to store the result of calling `set` member function **`insert`**. The `pair` contains an **`iterator`** pointing to the item in the `set` and a `bool` indicating whether the item was inserted—`true` if the item was not previously in the `set` and `false` otherwise. In this case, line 19 uses function **`insert`** to place the value `13.8` in the `set` and returns a `pair` in which `p.first` points to the value `13.8` in the `set` and `p.second` is `true` because the value was inserted.

```
16     // insert 13.8 in doubles; insert returns pair in which
17     // p.first represents location of 13.8 in doubles and
18     // p.second represents whether 13.8 was inserted
19     auto p{doubles.insert(13.8)}; // value not in set
20     std::cout << fmt::format("\n{} {} inserted\n", *(p.first),
21        (p.second ? "was" : "was not"));
22     std::cout << "doubles contains: ";
23     std::ranges::copy(doubles, output);
24
13.8 was inserted
doubles contains: 2.1 3.7 4.2 9.5 13.8
```

Inserting an Existing Value into a `set`

Line 26 attempts to insert `9.5`, which is already in the `set`. The output shows that `9.5` was not inserted because `set`s don't allow duplicate keys. In this case, `p.first` in the returned `pair` points to the existing `9.5` in the `set` and `p.second` is `false`.

```
25      // insert 9.5 in doubles
26      p = doubles.insert(9.5); // value already in set
27      std::cout << fmt::format("\n{} {} inserted\n", *(p.first),
28         (p.second ? "was" : "was not"));
29      std::cout << "doubles contains: ";
30      std::ranges::copy(doubles, output);
31      std::cout << "\n";
32   }
```

```
9.5 was not inserted
doubles contains: 2.1 3.7 4.2 9.5 13.8
```

13.9.3 `multimap` Associative Container

The **`multimap` associative container** is used for fast storage and retrieval of keys and associated values (often called **key–value pairs**). Many of the functions used with **`multiset`s** and **`set`s** are also used with **`multimap`s** and **`map`s**. The elements of `multimap`s and `map`s are `pair`s of keys and values. When inserting into a `multimap` or `map`, you use a `pair` object containing the key and the value. The ordering of the keys is determined by a **comparator function object**. For example, in a `multimap` that uses integers as the key type, keys can be sorted in **ascending order** by ordering them with **comparator function object `less<int>`**.

Duplicate keys are allowed in a `multimap`, so multiple values can be associated with a single key. This is called a **one-to-many relationship**. For example, in a credit-card transaction-processing system, one credit-card account can have many associated transactions; in a university, one student can take many courses, and one professor can teach many students; in the military, one rank (like "private") has many people. A **`multimap`** supports **bidirectional iterators**.

Creating a `multimap` Containing Key–Value Pairs

Perf Figure 13.8 demonstrates the **`multimap` associative container** (header **`<map>`**). If the order of the keys is not important, you can use **`unordered_multimap`** (header `<unordered_map>`) instead. A `multimap` is implemented to efficiently locate all values paired with a given key. Line 8 creates a `multimap` in which the key type is `int`, the type of a key's associated value is `double`, and the elements are ordered in ascending order by default. This is equivalent to

```
std::multimap<int, double, std::less<int>> pairs{};
```

```
1  // fig13_08.cpp
2  // Standard library multimap class template.
3  #include <fmt/format.h> // C++20: This will be #include <format>
4  #include <iostream>
5  #include <map> // multimap class-template definition
6
7  int main() {
8     std::multimap<int, double> pairs{}; // create multimap
```

Fig. 13.8 Standard library `multimap` class template.

Counting the Number of Key–Value Pairs for a Specific Key

Line 10 uses member function **`count`** to determine the number of key–value pairs with a key of `15`—in this case, `0` since the container is currently empty.

```
 9     std::cout << fmt::format("Number of 15 keys in pairs: {}\n",
10     pairs.count(15));
11
```

```
Number of 15 keys in pairs: 0
```

Inserting Key–Value Pairs

Line 13 uses member function `insert` to add a new key–value pair to the `multimap`. Standard library function `make_pair` creates a `pair` object, inferring the types of its arguments. In this case, `first` represents a key (`15`) of type `int` and `second` represents a value (`99.3`) of type `double`. Line 14 inserts another `pair` object with the key `15` and the value `2.7`. Then lines 15–16 output the number of `pair`s with key `15`.

```
12     // insert two pairs
13     pairs.insert(std::make_pair(15, 99.3));
14     pairs.insert(std::make_pair(15, 2.7));
15     std::cout << fmt::format("Number of 15 keys in pairs: {}\n\n",
16        pairs.count(15));
17
```

```
Number of 15 keys in pairs: 2
```

Inserting Key–Value Pairs with Braced Initializers Rather Than `make_pair`

You can use **braced initialization** for `pair` objects, so lines 13–14 can be simplified as

```
pairs.insert({15, 99.3});
pairs.insert({15, 2.7});
```

Lines 19–23 insert five additional `pair`s into the `multimap`. The range-based `for` statement in lines 28–30 outputs the `multimap`'s keys and values. We infer the type of the loop's control variable—in this case, a `pair` containing an `int` key and a `double` value. Line 29 accesses each `pair`'s members. Notice that the keys appear in ascending order because the `multimap` maintains the keys in ascending order.

```
18     // insert five pairs
19     pairs.insert({30, 111.11});
20     pairs.insert({10, 22.22});
21     pairs.insert({25, 33.333});
22     pairs.insert({20, 9.345});
23     pairs.insert({5, 77.54});
24
25     std::cout << "Multimap pairs contains:\nKey\tValue\n";
26
27     // walk through elements of pairs
28     for (const auto& mapItem : pairs) {
29        std::cout << fmt::format("{}\t{}\n", mapItem.first, mapItem.second);
30     }
31  }
```

```
Multimap pairs contains:
Key     Value
5       77.54
10      22.22
15      99.3
15      2.7
20      9.345
25      33.333
```

```
30      111.11
```

11 C++11: **Brace Initializing a Key–Value Pair Container**

This example used separate calls to member function `insert` to place key–value pairs in a `multimap`. If you know the key–value pairs in advance, you can use braced initialization when creating the `multimap`. For example, the following statement initializes a `multimap` with three key–value `pair`s that are represented by the sublists in the main initializer list:

```
std::multimap<int, double> pairs{
   {10, 22.22}, {20, 9.345}, {5, 77.54}};
```

13.9.4 `map` Associative Container

The **`map` associative container** (from **header `<map>`**) performs fast storage and retrieval of unique keys and associated values. Duplicate keys are not allowed—a single value can be associated with each key. This is called a **one-to-one mapping**. For example, a company that uses unique employee numbers, such as 100, 200 and 300, might have a `map` that associates employee numbers with their telephone extensions—4321, 4115 and 5217, respectively. Providing the key in a `map`'s subscript operator `[]` locates the value associated with that key in the `map`. If the order of the keys is not important, consider an **`unordered_map`** (**header `<unordered_map>`**), but keep in mind that the keys must be hashable.

Figure 13.9 demonstrates a `map` (lines 9–10). Only six of the initial eight key–value pairs are inserted because two have duplicate keys. Unlike similar data structures in some programming languages, inserting two key–value pairs with the same key does not replace the first key's value with that of the second. If you insert a key–value pair for which the key is already in the `map`, the key–value pair is ignored.

```
1  // fig13_09.cpp
2  // Standard library class map class template.
3  #include <iostream>
4  #include <fmt/format.h> // C++20: This will be #include <format>
5  #include <map> // map class-template definition
6
7  int main() {
8     // create a map; duplicate keys are ignored
9     std::map<int, double> pairs{{15, 2.7}, {30, 111.11}, {5, 1010.1},
10       {10, 22.22}, {25, 33.333}, {5, 77.54}, {20, 9.345}, {15, 99.3}};
11
12    // walk through elements of pairs
13    std::cout << "pairs contains:\nKey\tValue\n";
14    for (const auto& pair : pairs) {
15       std::cout << fmt::format("{}\t{}\n", pair.first, pair.second);
16    }
17
18    pairs[25] = 9999.99; // use subscripting to change value for key 25
19    pairs[40] = 8765.43; // use subscripting to insert value for key 40
20
21    // walk through elements of pairs
22    std::cout << "\nAfter updates, pairs contains:\nKey\tValue\n";
```

```
23     for (const auto& pair : pairs) {
24        std::cout << fmt::format("{}\t{}\n", pair.first, pair.second);
25     }
26  }
```

```
pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      33.333
30      111.11

After updates, pairs contains:
Key     Value
5       1010.1
10      22.22
15      2.7
20      9.345
25      9999.99
30      111.11
40      8765.43
```

Fig. 13.9 Standard library `map` class template.

Lines 18–19 use the `map` subscript operator, `[]`. When the subscript is a key that's in the `map`, the operator returns a reference to the associated value. **When the subscript is a key that's not in the `map`, the subscript operator inserts a new key–value pair in the `map`, consisting of the specified key and the default value for the container's value type**. Line 18 replaces the value for the key `25` (previously `33.333`, as specified in line 10) with a new value, `9999.99`. Line 19 inserts a new key–value `pair` in the `map`.

13.10 Container Adaptors

The three container adaptors are `stack`, `queue` and `priority_queue`. Container adaptors do not provide a data-structure implementation in which elements can be stored and **do not support iterators**. With an adaptor class, you can choose the underlying sequence container or use the adaptor's default choice—`deque` for `stack`s and `queue`s, or `vector` for `priority_queue`. The adaptor classes provide member functions `push` and `pop`:

- `push` properly inserts an element into an adaptor's underlying container.
- `pop` properly removes an element from an adaptor's underlying container.

Let's see examples of the adaptor classes.

13.10.1 `stack` Adaptor

Class `stack` (from header `<stack>`) enables insertions into and deletions from the underlying container at one end (the **top**). So, a `stack` is commonly referred to as a **last-in, first-out** data structure. A `stack` can be implemented with a `vector`, `list` or `deque`. By default, a `stack` is implemented with a `deque`.[19] The `stack` operations are

19. "`std::stack`." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/stack.

- `push` to insert an element at the `stack`'s **top**—implemented by calling the underlying container's `push_back` member function,
- `emplace` to construct an element in place at the `stack`'s **top**,

- `pop` to remove the `stack`'s **top** element—implemented by calling the underlying container's `pop_back` member function,
- `top` to get a reference to the `stack`'s top element—implemented by calling the underlying container's `back` member function,
- `empty` to determine whether the `stack` is empty—implemented by calling the underlying container's `empty` member function, and
- `size` to get the `stack`'s number of elements—implemented by calling the underlying container's `size` member function.

Figure 13.10 demonstrates `stack`. This example creates three `stack`s of `int`s, using a `deque` (line 26), a `vector` (line 27) and a `list` (line 28) as the underlying data structure.

```
1  // fig13_10.cpp
2  // Standard library stack adaptor class.
3  #include <iostream>
4  #include <list> // list class-template definition
5  #include <ranges>
6  #include <stack> // stack adaptor definition
7  #include <vector> // vector class-template definition
8
9  // pushElements generic lambda to push values onto a stack
10 auto pushElements = [](auto& stack) {
11    for (auto i : std::views::iota(0, 10)) {
12       stack.push(i); // push element onto stack
13       std::cout << stack.top() << ' '; // view (and display) top element
14    }
15 };
16
17 // popElements generic lambda to pop elements off a stack
18 auto popElements = [](auto& stack) {
19    while (!stack.empty()) {
20       std::cout << stack.top() << ' '; // view (and display) top element
21       stack.pop(); // remove top element
22    }
23 };
24
25 int main() {
26    std::stack<int> dequeStack{}; // uses a deque by default
27    std::stack<int, std::vector<int>> vectorStack{}; // use a vector
28    std::stack<int, std::list<int>> listStack{}; // use a list
29
30    // push the values 0-9 onto each stack
31    std::cout << "Pushing onto dequeStack: ";
32    pushElements(dequeStack);
33    std::cout << "\nPushing onto vectorStack: ";
34    pushElements(vectorStack);
35    std::cout << "\nPushing onto listStack: ";
36    pushElements(listStack);
37
38    // display and remove elements from each stack
39    std::cout << "\n\nPopping from dequeStack: ";
40    popElements(dequeStack);
41    std::cout << "\nPopping from vectorStack: ";
```

```
42     popElements(vectorStack);
43     std::cout << "\nPopping from listStack: ";
44     popElements(listStack);
45     std::cout << "\n";
46  }
```

```
Pushing onto dequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto vectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto listStack: 0 1 2 3 4 5 6 7 8 9

Popping from dequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from vectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from listStack: 9 8 7 6 5 4 3 2 1 0
```

Fig. 13.10 Standard library `stack` adaptor class.

The generic lambda `pushElements` (lines 10–15) pushes 0–9 onto a `stack`. Lines 32, 34 and 36 call this lambda for each `stack`. Line 12 uses function **`push`** (available in each adaptor class) to place an integer on top of the `stack`. Line 13 uses `stack` function **`top`** to retrieve the `stack`'s top element for output. Function **`top`** does not remove the top element.

The generic lambda `popElements` (lines 18–23) pops the elements off a `stack`. Lines 40, 42 and 44 call this lambda for each `stack`. Line 20 uses `stack` function **`top`** to retrieve the `stack`'s **top** element for output. Line 21 uses function **`pop`**, available in each adaptor class, to remove the `stack`'s top element. Function **`pop`** does not return a value. So, you must call **`top`** to obtain the top element's value before you **`pop`** that element from the stack.

13.10.2 `queue` Adaptor

A queue is similar to a waiting line. The first in line is the first removed. So, a queue is referred to as a **first-in, first-out (FIFO)** data structure. Class **`queue`** (from header **`<queue>`**) enables insertions only at the **back** of the underlying data structure and deletions only from the **front**. A `queue` stores its elements in objects of the **`list`** or **`deque`** sequence containers—**`deque`** is the default.[20] The common `queue` operations are

20. "`queue`." Accessed January 22, 2022. https://en.cppreference.com/w/cpp/container/queue.

- **`push`** to insert an element at the `queue`'s back—implemented by calling the underlying container's **`push_back`** member function,
- **`emplace`** to construct an element in place at the `queue`'s **top**,
- **`pop`** to remove the element at the `queue`'s front—implemented by calling the underlying container's **`pop_front`** member function,
- **`front`** to get a reference to the `queue`'s first element—implemented by calling the underlying container's **`front`** member function,
- **`back`** to get a reference to the `queue`'s last element—implemented by calling the underlying container's **`back`** member function,
- **`empty`** to determine whether the `queue` is empty—this calls the underlying container's **`empty`** member function, and
- **`size`** to get the `queue`'s number of elements—this calls the underlying container's **`size`** member function.

Figure 13.11 demonstrates the **`queue`** adaptor class. Line 7 instantiates a `queue` of `double`s. Lines 10–12 use function **`push`** to add elements to the `queue`. The `while` statement in lines 17–20 uses function **`empty`** (available in all containers) to determine whether the `queue` is empty (line 17). While there are more elements in the `queue`, line 18 uses `queue` function **`front`** to read (but not remove)

the queue's first element for output. Line 19 removes the queue's first element with function pop, available in all adaptor classes.

```
1  // fig13_11.cpp
2  // Standard library queue adaptor class template.
3  #include <iostream>
4  #include <queue> // queue adaptor definition
5
6  int main() {
7     std::queue<double> values{}; // queue with doubles
8
9     // push elements onto queue values
10    values.push(3.2);
11    values.push(9.8);
12    values.push(5.4);
13
14    std::cout << "Popping from values: ";
15
16    // pop elements from queue
17    while (!values.empty()) {
18       std::cout << values.front() << ' '; // view front element
19       values.pop(); // remove element
20    }
21
22    std::cout << "\n";
23 }
```

```
Popping from values: 3.2 9.8 5.4
```

Fig. 13.11 Standard library queue adaptor class template.

13.10.3 priority_queue Adaptor

Class **priority_queue** (from header <queue>) enables insertions in sorted order into the underlying data structure and deletions from the front of the underlying data structure. By default, a **priority_queue** stores its elements in a **vector**.[21] Elements added to a **prior-ity_queue** are inserted in **priority order**, such that the highest-priority element will be the first element removed. This is usually accomplished by arranging the elements as a **heap**[22] that always maintains its highest-priority element at the front of the data structure. Element comparisons are performed with **comparator function object** **less<T>** by default.

21. "std::priority_queue." Access April 15, 2021. https://en.cppreference.com/w/cpp/container/priority_queue.

22. Not to be confused with the heap for dynamically allocated memory. See Section 14.4.14.

The **priority_queue** operations include

- **push** to insert an element at the appropriate location based on **priority order**.
- **emplace** to construct an element in place and re-sort the priority_queue into **priority order**,
- **pop** to remove the priority_queue's **highest-priority** element.
- **top** to get a reference to the priority_queue's **top** element—implemented by calling the underlying container's **front** member function.
- **empty** to determine whether the priority_queue is empty—implemented by calling the underlying container's **empty** member function.
- **size** to get the priority_queue's number of elements—implemented by calling the underlying container's **size** member function.

Figure 13.12 demonstrates `priority_queue`. Line 7 instantiates a `priority_queue` of `double`s and uses a **`vector`** as the underlying data structure. Lines 10–12 use member function **`push`** to add elements to the `priority_queue`. Lines 17–20 use member function `empty` (available in all containers) to determine whether the `priority_queue` is empty (line 17). If not, line 18 uses `priority_queue` function **`top`** to retrieve the **highest-priority** element (i.e., the largest value) in the `priority_queue` for output. Line 19 calls **`pop`**, available in all adaptor classes, to remove the `priority_queue`'s **highest-priority** element.

```
1  // fig13_12.cpp
2  // Standard library priority_queue adaptor class.
3  #include <iostream>
4  #include <queue> // priority_queue adaptor definition
5
6  int main() {
7     std::priority_queue<double> priorities; // create priority_queue
8
9     // push elements onto priorities
10    priorities.push(3.2);
11    priorities.push(9.8);
12    priorities.push(5.4);
13
14    std::cout << "Popping from priorities: ";
15
16    // pop element from priority_queue
17    while (!priorities.empty()) {
18       std::cout << priorities.top() << ' '; // view top element
19       priorities.pop(); // remove top element
20    }
21
22    std::cout << "\n";
23 }
```

```
Popping from priorities: 9.8 5.4 3.2
```

Fig. 13.12 Standard library `priority_queue` adaptor class.

13.11 `bitset` Near Container

Class **`bitset`** (header **`<bitset>`**) makes it easy to create and manipulate **bit sets**, which are useful for representing a set of bit flags. `bitset`s are fixed in size at compile-time. Class `bitset` is an alternative tool for bit manipulation, discussed in online Appendix E.

The declaration

```
bitset<size> b;
```

creates `bitset b`, in which every one of the `size` bits is initially `0` ("off").

The statement

```
b.set(bitNumber);
```

sets bit `bitNumber` "on." You also can call this function with a `bool` second argument specifying whether to set the bit "on" (`true`) or "off" (`false`). The expression `b.set()` sets all bits in `b` "on."

The statement

```
b.reset(bitNumber);
```

sets bit `bitNumber` “off.” The expression `b.reset()` sets all bits in `b` “off.”

The statement

```
b.flip(bitNumber);
```

toggles bit `bitNumber`—if the bit is “on,” `flip` sets it “off,” and vice versa. The expression `b.flip()` flips all bits in `b`.

The expression

```
b[bitNumber]
```

for a non-`const bitset` returns a `std::bitset::reference`[23] that enables you to manipulate the `bool` at position `bitNumber`; otherwise, it returns a copy of the `bool`.

23. “`std::bitset<N>::reference`.” Accessed January 23, 2022. https://en.cppreference.com/w/cpp/utility/bitset/reference.

The expression

```
b.test(bitNumber);
```

performs range checking on `bitNumber` first. If `bitNumber` is in range (based on the number of bits in the `bitset`), `test` returns `true` if the bit is on or `false` if it’s off. Otherwise, `test` throws an `out_of_range` exception.

The expression

```
b.size()
```

returns the number of bits.

The expression

```
b.count()
```

returns the number of bits that are set (`true`).

The expression

```
b.any()
```

11 (added in C++11) returns `true` if any bit is set.

The expression

```
b.all()
```

11 (added in C++11) returns `true` if all of the bits are set (`true`).

The expression

```
b.none()
```

11 (added in C++11) returns `true` if none of the bits is set (that is, all the bits are `false`).

The expressions

```
b == b1
b != b1
```

compare the two `bitset`s for equality and inequality, respectively.

Each of the bitwise assignment operators `&=`, `|=` and `^=` (discussed in detail in online Appendix E) can be used to combine `bitset`s. For example,

```
b &= b1;
```

performs a bit-by-bit AND between `b` and `b1`, setting each bit in `b` "on" if it's "on" in both `b` and `b1`. Bitwise OR

```
b |= b1;
```

sets each bit in `b` "on" if it's "on" in either or both of `b` and `b1`. Bitwise XOR

```
b ^= b2;
```

sets each bit in `b` "on" if it's "on" in only `b` and `b1`, but not both.

You also can perform a bitwise NOT operation. The following expression returns a copy of the `bitset` with all its bits flipped:

```
~b
```

The statement

```
b >>= n;
```

shifts the bits right by `n` positions. The expression

```
b <<= n;
```

shifts the bits left by `n` positions.

The expressions

```
b.to_string()
b.to_ulong()
b.to_ullong()
```

convert the `bitset` to a `string`, an `unsigned long` or an `unsigned long long`, respectively.

13.12 Optional: A Brief Intro to Big *O*

In this chapter, you've seen Big *O* notations of $O(1)$, $O(n)$ and $O(\log n)$. In this section, we'll explain what those mean and introduce two others—$O(n^2)$ and $O(n \log n)$. These Big *O* expressions characterize the amount of work an algorithm must do when processing n items. On today's desktop

computers, which may process a few billion operations per second, that translates to whether a program will run almost instantaneously or take seconds, minutes, hours, days, months, years or even more to complete. Obviously, you'd prefer algorithms that complete quickly, even though the number of items n that they may be processing is large. This is especially true in today's world of Big Data computing applications. In the implementation of the standard library containers and algorithms, Big Os of $O(1)$, $O(n)$, $O(\log n)$ and even $O(n \log n)$—which is common for relatively good sorting algorithms—are considered to be reasonably efficient. Algorithms categorized by $O(n^2)$ or worse, such as $O(2^n)$ or $O(n!)$, could run on a modest number of items for centuries, millennia or longer. So you'll want to avoid writing such algorithms.

Searching algorithms all accomplish the same goal—finding in a container (which for this discussion we'll assume is an array container) an element matching a given search key, if such an element does, in fact, exist. There are, however, several things that differentiate search algorithms from one another. **The major difference is the amount of effort they require to complete the search, based on the number of items they must search through.** One way to describe this effort is with Big O notation, which indicates how hard an algorithm may have to work to solve a problem. For searching and sorting algorithms, this depends mainly on how many data elements there are.

$O(1)$ Algorithms

Suppose an algorithm tests whether the first element of an array is equal to the second. If the array has 10 elements, this algorithm requires one comparison. If the array has 1,000 elements, it still requires one comparison. In fact, the algorithm is independent of the number, n, of elements in the array. This algorithm is said to have constant running time, which is represented in Big O notation as $O(1)$ and pronounced as "order one." An algorithm that's $O(1)$ does not necessarily require only one comparison. $O(1)$ just means that the number of comparisons is *constant*—it does not grow as the size of the array increases. An algorithm that tests whether the first element of an array is equal to any of the next three elements is still $O(1)$ even though it requires three comparisons.

$O(n)$ Algorithms

An algorithm that tests whether the first array element is equal to *any* of the other array elements will require at most $n - 1$ comparisons, where n is the number of array elements. If the array has 10 elements, this algorithm requires up to nine comparisons. If the array has 1,000 elements, it requires up to 999 comparisons. As n grows larger, the n part of the expression $n - 1$ "dominates," and subtracting one becomes inconsequential. Big O is designed to highlight these dominant terms and ignore terms that become unimportant as n grows. For this reason, an algorithm that requires a total of $n - 1$ comparisons (such as the one we described earlier) is said to be $O(n)$. An $O(n)$ algorithm is referred to as having a linear running time. $O(n)$ is pronounced "on the order of n" or simply "order n."

$O(n^2)$ Algorithms

Now suppose you have an algorithm that tests whether *any* element of an array is duplicated elsewhere in the array. The first element must be compared with all other elements in the array. The second element must be compared with all elements except the first (it was already compared to the first). The third element must be compared with all elements except the first two. In the end, this algorithm makes $(n - 1) + (n - 2) + \ldots + 2 + 1$ or $n^2/2 - n/2$ comparisons. As n increases, the n^2 term dominates, and the n term becomes inconsequential. Again, Big O notation highlights the n^2 term, ignoring $n/2$.

Big O is concerned with how an algorithm's running time grows in relation to the number of items processed. Suppose an algorithm requires n^2 comparisons. With four elements, the algorithm requires 16 comparisons; with eight elements, 64 comparisons. With this algorithm, *doubling* the number of elements *quadruples* the number of comparisons. Consider a similar algorithm requiring $n^2/2$ comparisons. With four elements, the algorithm requires eight comparisons; with eight elements, 32 comparisons. Again, *doubling* the number of elements *quadruples* the number of comparisons. Both of these algorithms grow as the square of n, so Big O ignores the constant, and both algorithms are considered to be $O(n^2)$, which is referred to as **quadratic running time** and pronounced "on the order of n-squared" or more simply "order n-squared."

Efficiency of the Linear Search $O(n)$

The linear search algorithm, which is typically used for searching an unsorted array, runs in $O(n)$ time. The worst case in this algorithm is that every element must be checked to determine whether the search key exists in the array. If the size of the array is doubled, the number of comparisons that the algorithm must perform is also doubled. Linear search can provide outstanding performance if the element matching the search key happens to be at or near the front of the array. But we seek algorithms that perform well, on average, across all searches, including those where the element matching the search key is near the end of the array.

Linear search is easy to program, but it can be slow compared to other search algorithms, especially as n gets large. If a program needs to perform many searches on large arrays, it's better to implement a more efficient algorithm, such as the `binary_search` algorithm, which we'll use in Chapter 14.

Perf Sometimes the simplest algorithms perform poorly. Their virtue often is that they're easy to program, test and debug. Sometimes more complex algorithms are required to realize maximum performance.

Efficiency of the Binary Search $O(\log n)$

In the worst-case scenario, searching a *sorted* array of 1,023 elements ($2^{10} - 1$) takes *only 10 comparisons* with binary search. The algorithm compares the search key to the middle element of the sorted array. If it's a match, the search is over. More likely, the search key will be larger or smaller than the middle element. If it's larger, we can eliminate the array's first half from consideration. If it's smaller than the middle element, we can eliminate the array's second half from consideration. This creates a halving effect so that on subsequent searches, we have to search only 511, 255, 127, 63, 31, 15, 7, 3 and 1 elements. The number 1,023 ($2^{10} - 1$) needs to be halved only 10 times to either find the key or determine that it's not in the array. Dividing by 2 is equivalent to one comparison in the binary search algorithm. Thus, an array of 1,048,575 ($2^{20} - 1$) elements takes a *maximum of 20 comparisons* to find the key, and an array of about one billion elements takes a *maximum of 30 comparisons* to find the key. This is a tremendous improvement in performance over the linear search. For a one-billion-element array, this is a difference between an average of 500 million comparisons for the linear search and a maximum of only 30 comparisons for the binary search! The maximum number of comparisons needed for the binary search of any sorted array is the exponent of the first power of 2 greater than the number of elements in the array, which is represented as $\log_2 n$. All logarithms grow at "roughly the same rate," so for Big O comparison purposes, the base can be omitted. This results in a Big O of $O(\log n)$ for a binary search, which is also known as **logarithmic running time** and pronounced as "order log n."

Common Big *O* Notations

The following table lists various common Big *O* notations and several values for *n* to highlight the differences in the growth rates. If you interpret the values in the table as seconds of calculation, you can easily see why $O(n^2)$ algorithms are to be avoided!

n	*O*(1	*O*(log *n*)	*O*(*n*)	*O*(*n* log *n*)	$O(n^2)$
1	1	0	1	0	1
2	1	1	2	2	4
3	1	1	3	3	9
4	1	1	4	4	16
5	1	1	5	5	25
10	1	1	10	10	100
100	1	2	100	200	10,000
1,000	1	3	1,000	3,000	10^6
1,000,000	1	6	1,000,000	6,000,000	10^{12}

13.13 Optional: A Brief Intro to Hash Tables

When a program creates objects, it may need to store and retrieve them efficiently. Storing and retrieving information with arrays is efficient if some aspect of your data directly matches a numerical key value and if the *keys are unique* and tightly packed. If you have 100 employees with nine-digit Social Security numbers and you want to store and retrieve employee data by using the Social Security number as an array index, the task will require an array with over 800 million elements because nine-digit Social Security numbers must begin with 001–899 (excluding 666) as per the Social Security Administration's website:

```
https://www.ssa.gov/employer/randomization.html
```

This is impractical for most applications that use Social Security numbers as keys. A program having so large an array could achieve high performance for both storing and retrieving employee records by simply using the Social Security number as the array index.

Numerous applications have this problem—namely, that either the keys are of the wrong type (e.g., not positive integers usable as array subscripts) or they're of the right type, but sparsely spread over a huge range, such as Social Security numbers for a small company's employees. What is needed is a high-speed scheme for converting keys such as Social Security numbers, inventory part numbers and the like into unique array indices over a modest-size array. Then, when an application needs to store a data item, the scheme can convert the application's key rapidly into an array index (a process called **hashing**), and the item can be stored at that slot in the array. Retrieval is accomplished the same way: Once the application has a key for which it wants to retrieve the data, it simply applies the conversion to the key (again, called hashing)—this produces the array index where the data is stored and retrieved.

Why the name hashing? When we convert a key into an array index, we literally scramble the bits, forming a kind of "mishmashed," or hashed, number. The number actually has no real significance beyond its usefulness in storing and retrieving particular data.

A glitch in the scheme is called a **collision**—this occurs when two different keys "hash into" the same cell (or element) in the array. We cannot store two values in the same space, so we need to find an alternative home for all values beyond the first that hash to the same array index. There are many schemes for doing this. One is to "hash again"—that is, to apply another hashing transformation to the previous hash result to provide the next candidate cell in the array. The hashing process is designed to distribute the values throughout the table, so hopefully an available cell will be found with one or a few hashes.

Another scheme uses one hash to locate the first candidate cell. If that cell is occupied, adjacent cells are searched sequentially until an available cell is found. Retrieval works the same way: The key is hashed once to determine the initial location and check whether it contains the desired data. If it does, the search is finished; otherwise, successive cells are searched sequentially until the desired data is found. A popular solution to hash-table collisions is to have each cell of the table be a **hash bucket**—typically a linked list of all the key–value pairs that hash to that cell.

A hash table's **load factor** affects the performance of hashing schemes. The load factor is the ratio of the number of occupied cells in the hash table to the total number of cells in the hash table. The closer this ratio gets to 1.0, the greater the chance of collisions, which slow data insertion and retrieval.

Perf The load factor in a hash table is a classic example of a memory-space/execution-time trade-off: By increasing the load factor, we get better memory utilization, but the program runs slower due to increased hashing collisions. By decreasing the load factor, we get better program speed, because of reduced hashing collisions, but we get poorer memory utilization because a larger portion of the hash table remains empty.

C++'s `unordered_set`, `unordered_multiset`, `unordered_map` and `unordered_multimap` associative containers are implemented as hash tables "under the hood." When you use those containers, you get the benefit of high-speed data storage and retrieval without having to build your own hash-table mechanisms—a classic example of reuse.

Another interesting application of hashing is with virtual memory.[24] The company VMWare[25] builds virtualization products. For example, on our macOS systems, we use VMWare

Fusion to run Windows and Linux in parallel with macOS. These are all virtual memory operating systems. Programs and data are maintained on massive secondary storage and brought into more limited primary memory (and even more limited and faster cache memory) to execute. The trick to running applications efficiently is to keep in primary memory (and cache memory) the portion of the program that needs to be executing at a given time. Finding those program pieces in the virtual memory involves searching the very large data structures that define the virtual memory. VMWare is exploring hashing techniques to use in concert with hardware called translation look-aside buffers (TLBs) to speed the process of locating the needed virtual memory pieces.[26]

24. "Virtual Memory." Wikipedia. Wikimedia Foundation. Accessed January 24, 2022. https://en.wikipedia.org/wiki/Virtual_memory.
25. "VMWare: About Us." Accessed January 24, 2022. https://www.vmware.com/company.html.
26. Alex Conway, Rob Johnson, Jayneel Gandhi, et al., "Hash-Based Virtual Memory: Reducing the Costs of Virtual Memory Through Better Data Structures." VMWare. Accessed January 23, 2021. https://research.vmware.com/projects/hashed-p-o-t-a-t-o-e.

13.14 Wrap-Up

In this chapter, we introduced three key components of the standard library—containers, iterators and algorithms. You learned about the linear **sequence containers,** `array` (Chapter 6), `vector`, `deque`, `forward_list` and `list`, which all represent linear data structures. We discussed the nonlinear **associative containers,** `set`, `multiset`, `map` and `multi-map` and their unordered versions. You also saw that the **container adaptors** `stack`, **queue** and `priority_queue` can be used to restrict the operations of the sequence containers `vector`, `deque` and `list` for the purpose of implementing the specialized data structures represented by the container adaptors. You learned the categories of iterators and that each algorithm can be used with any container that supports the minimum iterator functionality the algorithm requires. We distinguished between common ranges and C++20 ranges and demonstrated the `std::ranges::copy` algorithm. You also learned the features of class `bitset`, which makes it easy to create and manipulate bit sets as a near container.

Chapter 14 continues our discussion of the standard library's containers, iterators and algorithms with a detailed treatment of algorithms. You'll see that C++20 concepts have been used extensively in the latest version of the standard library. We'll discuss the minimum iterator requirements that determine which containers can be used with each algorithm. We'll continue our discussion of lambda expressions and see that function pointers and function objects—instances of classes that overload the function-call (parentheses) operator—can be passed to algorithms. In Chapter 15, you'll see how C++20 concepts come into play when we build a custom container and a custom iterator.

14. Standard Library Algorithms and C++20 Ranges & Views

Objectives

In this chapter, you'll:

- Understand minimum iterator requirements for working with standard library containers and algorithms.
- Create lambda expressions that capture local variables to use in the lambda expressions' bodies.
- Use many of the C++20 `std::ranges` algorithms.
- Understand the C++20 concepts corresponding to the C++20 `std::ranges` algorithms' minimum iterator requirements.
- Compare the new C++20 `std::ranges` algorithms with older common-range `std` algorithms.
- Use iterators with algorithms to access and manipulate the elements of standard library containers.
- Pass lambdas, function pointers and function objects into standard library algorithms mostly interchangeably.
- Use projections to transform objects in a range while processing them with C++20 range algorithms.
- Use C++20 views and lazy evaluation with C++20 ranges.
- Learn about C++ ranges features possibly coming in C++23.
- Be introduced to parallel algorithms for performance enhancement—we'll discuss these in Chapter 17.

Outline

14.1 Introduction
14.2 Algorithm Requirements: C++20 Concepts
14.3 Lambdas and Algorithms
14.4 Algorithms
14.4.1 `fill`, `fill_n`, `generate` and `generate_n`
14.4.2 `equal`, `mismatch` and `lexicographical_compare`
14.4.3 `remove`, `remove_if`, `remove_copy` and `remove_copy_if`
14.4.4 `replace`, `replace_if`, `replace_copy` and `replace_copy_if`
14.4.5 Shuffling, Counting, and Minimum and Maximum Element Algorithms
14.4.6 Searching and Sorting Algorithms
14.4.7 `swap`, `iter_swap` and `swap_ranges`
14.4.8 `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n`
14.4.9 `inplace_merge`, `unique_copy` and `reverse_copy`
14.4.10 Set Operations
14.4.11 `lower_bound`, `upper_bound` and `equal_range`
14.4.12 `min`, `max` and `minmax`
14.4.13 Algorithms `gcd`, `lcm`, `iota`, `reduce` and `partial_sum` from Header `<numeric>`
14.4.14 Heapsort and Priority Queues
14.5 Function Objects (Functors)
14.6 Projections
14.7 C++20 Views and Functional-Style Programming

14.7.1 Range Adaptors
14.7.2 Working with Range Adaptors and Views
14.8 Intro to Parallel Algorithms
14.9 Standard Library Algorithm Summary
14.10 A Look Ahead to C++23 Ranges
14.11 Wrap-Up

14.1 Introduction

This chapter is intended as a reference to the large number of standard library algorithms. It focuses on common container manipulations, including **filling with values**, **generating values**, **comparing elements or entire containers**, **removing elements**, **replacing elements**, **mathematical operations**, **searching**, **sorting**, **swapping**, **copying**, **merging**, **set operations**, **determining boundaries**, and **calculating minimums and maximums**. The standard library provides many prepackaged, templatized algorithms:

- 20 90 in the `<algorithm>` header's `std` namespace—82 also are overloaded in the **C++20** `std::ranges` **namespace**,
- 11 in the `<numeric>` header's `std` namespace, and
- 20 14 in the `<memory>` header's `std` namespace—all 14 also are overloaded in the **C++20** `std::ranges` **namespace**.

11 17 20 Many algorithms were added in C++11 and C++20, and a few in C++17. For the complete list of algorithms and their descriptions, visit

```
https://en.cppreference.com/w/cpp/algorithm
```

20 Minimum Algorithm Requirements and C++20 Concepts

Concepts The standard library's algorithms specify **minimum requirements** that help you determine which **containers**, **iterators** and **functions** can be passed to each algorithm. The **C++20 range-based algorithms** in the `std::ranges` **namespace** specify their requirements with **C++20 concepts**. We briefly introduce C++20 concepts as needed for you to understand the requirements for working with these algorithms. We'll discuss concepts in more depth in Chapter 15 as we build templates.

C++20 Range-Based Algorithms vs. Earlier Common-Range Algorithms

Most of the algorithms we present in this chapter have overloads in

- 20 the `std::ranges` **namespace** for use with **C++20 ranges**, and
- the `std` **namespace** for use with **pre-C++20 common ranges**—pairs of iterators representing a range's first element and the element one past the range's end.

We'll focus mainly on the C++20 ranges versions.[1,2]

1. Tristan Brindle, "An Overview of Standard Ranges," September 29, 2019. Accessed January 29, 2022. https://www.youtube.com/watch?v=SYLgG7Q5Zws.

2. Tristan Brindle, “C++20 Ranges in Practice,” October 8, 2020. Accessed January 29, 2022. https://www.youtube.com/watch?v=d_E-VLyUnzc.

Lambdas, Function Pointers and Function Objects

Section 6.14.2 introduced **lambda expressions**. In Section 14.3, we'll revisit them and introduce additional details. As you'll see, various algorithms can receive a lambda, a function pointer or a function object as an argument. Most examples in this chapter use **lambdas** because they're convenient for expressing small tasks. In Section 14.5, you'll see that a **lambda expression** often is interchangeable with a **function pointer** or a **function object** (also called a **functor**). A function object's class overloads the `operator()` **function**, allowing the object's name to be used as a function name, as in *objectName*(*arguments*). **Throughout this chapter, when we say that an algorithm can receive a function as an argument, we mean that it can receive a function, a lambda or a function object.**

20 C++20 Views and Functional-Style Programming with Lazy Evaluation

Like many modern languages, C++ offers **functional-style programming** capabilities, which we began introducing in Section 6.14.3. In particular, we demonstrated **filter**, **map and reduce operations**. In Section 14.7, we'll continue our presentation of functional-style programming with C++20's new `<ranges>` **library**.

Parallel Algorithms

17 C++17 introduced new parallel overloads for 69 standard library algorithms in the header `<algorithm>`, enabling you to enhance program performance on **multi-core architectures**. Section 14.8 briefly overviews the parallel overloads. We enumerate the algorithms with parallel versions in Section 14.9's standard-library-algorithms summary tables. **Chapter 17, Parallel Algorithms and Concurrency: A High-Level View**, demonstrates several parallel algorithms. In that chapter, we'll use features from the `<chrono>` header to time standard library algorithms running sequentially on a single core and running in parallel on multiple cores so you can see the performance improvement.

23 Looking Ahead to Ranges in C++23

Some C++20 algorithms—including those in the `<numeric>` **header** and the parallel algorithms in the `<algorithm>` **header**—do not have `std::ranges` overloads. Section 14.10 overviews updates expected in C++23 and mentions the **open-source project** `ranges-next`,[3] which contains implementations for some proposed updates.

3. Corentin Jabot, “Ranges for C++23.” Accessed January 29, 2022. https://github.com/cor3ntin/rangesnext.

14.2 Algorithm Requirements: C++20 Concepts

20 SE The C++ standard library separates containers from the algorithms that manipulate the containers. Most algorithms operate on container elements indirectly via iterators. **This architecture makes it easier to write generic algorithms applicable various containers**—a strength of the standard library algorithms.

Container class templates and their corresponding iterator class templates typically reside in the same header. For example, the `<vector>` header contains the templates for class `vector` and its iterator class. A container internally creates objects of its iterator class and returns them via container member functions, such as `begin`, `end`, `cbegin` and `cend`.

Iterator Requirements

SE For maximum reuse, **each algorithm can operate on any container that meets the algorithm's minimum iterator requirements.**[4] For example, an algorithm requiring **forward iterators** can operate on any container that provides **at least forward iterators**. **All standard library algorithms can operate on `vector`s and `array`s** because they support **contiguous iterators** that provide every iterator operation discussed in Section 13.3.

4. Alexander Stepanov and Meng Lee, "The Standard Template Library, Section 2: Structure of the Library," October 31, 1995. Accessed January 29, 2022. http://stepanovpapers.com/STL/DOC.PDF.

Before C++20,

- each container's documentation mentioned the iterator level it supported, and
- each algorithm's documentation mentioned its minimum iterator requirements.

You were expected to adhere to an algorithm's documented requirements by using only containers with iterators that satisfied those requirements. If you passed the wrong iterator type, the compiler would substitute that type throughout the algorithm's template definition and, as Bjarne Stroustrup observed, produce "spectacularly bad error messages."[5]

5. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—1. A Bit of Background," January 31, 2017. Accessed January 29, 2022. http://wg21.link/p0557r0.

C++20 Concepts

One of the "big four" C++20 features is **concepts**—a technology for constraining the types used with templates. Stroustrup points out that "Concepts complete C++ templates as originally envisioned"[6] decades ago. Each concept specifies a type's requirements or a relationship between types.[7]

6. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—Conclusion," January 31, 2017. Accessed January 29, 2022. http://wg21.link/p0557r0.
7. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces," January 31, 2017. Accessed January 29, 2022. http://wg21.link/p0557r0.

When an algorithm's parameter is constrained by a concept, the compiler can check the requirements in the function call before substituting the argument's type throughout the function template. If your argument's type does not satisfy the concept's requirements, a benefit of concepts is that the compiler produces many fewer and much clearer error messages than for the older **common-range algorithms**. This makes it easier for you to understand the errors and correct your code.

20 This chapter's primary focus is **C++20's new range-based algorithms**, which are **constrained with many C++20 predefined concepts**. There are 76 predefined concepts in the standard.[8,9] Though we've used standard library templates extensively in Chapters 6 and 13 and will do so again here, we have not used concepts in the code, nor will we in this chapter. The programmers responsible for

creating **C++20's range-based algorithms** used concepts in the algorithms' function template signatures to specify the algorithms' iterator and range requirements.

---

8. "Index of Library Concepts." Accessed January 29, 2022. https://timsong-cpp.github.io/cppwp/n4861/conceptindex.
9. The standard also specifies 30 "exposition-only" concepts used only for discussion purposes. "Exposition-Only in the C++ Standard?" Answered December 28, 2015. Accessed January 29, 2022. https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard.

When invoking **C++20's range-based algorithms**, you must pass container and iterator arguments that meet the algorithms' requirements. So, for the algorithms we present in this chapter, we'll mention the predefined concept names specified in the algorithms' prototypes and briefly explain how they constrain the algorithms' arguments. We'll call out the concepts with the icon you see in the margin next to this paragraph. In Chapter 15, we'll take a template developer's viewpoint as we demonstrate implementing custom templates with concepts.

C++20 Iterator Concepts

As you view the **C++20 range-based algorithms**' documentation, you'll often see in their prototypes the following **C++20 iterator concepts**, which are defined in namespace `std` in the `<iterator>` **header**:

- `input_iterator`
- `output_iterator`
- `forward_iterator`
- `bidirectional_iterator`
- `random_access_iterator`
- `contiguous_iterator`

These specify the type requirements for the **iterator categories** introduced in Chapter 13. For a complete list of **iterator concepts**, see the "C++20 iterator concepts" section at

```
https://en.cppreference.com/w/cpp/iterator
```

C++20 Range Concepts

The `range` **concept** describes a type with a **begin iterator** and an **end sentinel**, possibly of different types. You'll often see the following **C++20 ranges concepts** in the **C++20 range-based algorithms**' prototypes:

- `input_range`—a range that supports `input_iterator`s
- `output_range`—a range that supports `ouput_iterator`s
- `forward_range`—a range that supports `forward_iterator`s
- `bidirectional_range`—a range that supports `bidirectional_iterator`s
- `random_access_range`—a range that supports `random_access_iterator`s
- `contiguous_range`—a range that supports `contiguous_iterator`s

These are defined in the `std::ranges` **namespace** in the `<ranges>` header. They specify the requirements for **ranges** supporting the **iterator categories** discussed in Chapter 13. We'll discuss other concepts as we encounter them in the coming chapters.

You can customize many standard library algorithms' behavior by passing a function as an argument. You saw in Section 6.14.2 that **lambda expressions define anonymous functions**—usually, inside other functions.[10] In Section 8.19, you saw that they can manipulate an enclosing function's local variables. We'll typically pass lambdas to standard library algorithms because they're convenient for expressing small tasks.

10. Lambdas also can be defined at class or namespace scope. For details, see https://en.cppreference.com/w/cpp/language/lambda.

Figure 14.1 revisits the standard library's `copy` and `for_each` **algorithms** using the **C++20's** `std::ranges` versions. Recall that `for_each` receives as one of its arguments a function specifying a task to perform on each container element.

```
1  // fig14_01.cpp
2  // Lambda expressions.
3  #include <algorithm>
4  #include <array>
5  #include <iostream>
6  #include <iterator>
7
8  int main() {
9     std::array values{1, 2, 3, 4}; // initialize values
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output);
14
values contains: 1 2 3 4
```

Fig. 14.1 Lambda expressions.

Algorithm `copy` and Common Ranges Iterator Requirements vs. C++20 Ranges Iterator Requirements

20 Line 9 creates an `array` of `int` values, which line 13 displays using an `ostream_iterator` and the **`copy` algorithm**[11] from **C++20's** `std::ranges` **namespace**. Section 13.6.2 called the **common ranges** `std::copy` algorithm as follows:

11. "`std::ranges::copy`, `std::ranges::copy_if`, `std::ranges::copy_result`, `std::rang-es::copy_if_result`." Accessed January 29, 2022. https://en.cppreference.com/w/cpp/algorithm/ranges/copy.

```
std::copy(integers.cbegin(), integers.cend(), output);
```

This algorithm's documentation states that the first two arguments must be **input iterators** designating the beginning and end of the **common range** to copy. The third argument was an **output iterator** indicating where to copy the elements.

Line 13 calls the `std::ranges` version of `copy` with just **two arguments**—the `values` container and an `ostream_iterator`. This version of `copy` determines `values`' beginning and end for you by calling

```
std::ranges::begin(values)
```

and

```
std::ranges::end(values)
```

Concepts **Any container that supports `begin` and `end` iterators can be treated as a C++20 range.** The first argument to the `std::ranges::copy` algorithm must be an `input_range`, and the second must be an **output iterator**. Here, we write to the standard output stream, but we also could write into another range. The **output iterator** has several requirements:

- Concepts It must be `std::weakly_incrementable`, meaning it must support the `++` operator to enable iteration through a sequence. All iterators support this operator.
- Concepts Concepts Concepts It also must be `std::indirectly_copyable`, which specifies a relationship between the iterator for `copy`'s `input_range` and the **output iterator**. In particular, the `input_range`'s iterator must be `std::indirectly_readable`, enabling `copy` to dereference the iterator to read an element of a given type, and the **output iterator** must be `std::indirectly_writable`, enabling `copy` to dereference the iterator to write the copied element of that type into the target range.

Concepts The concepts `indirectly_readable` and `indirectly_writable`, in turn, have many additional requirements. For simplicity going forward, we'll say **output iterator** when an algorithm requires a `weakly_incrementable` output iterator.

SE 20 Err **Use C++20's range-based algorithms rather than the older common-ranges algorithms. Passing an entire container, rather than begin and end iterators, simplifies your code and eliminates accidentally mismatched iterators—that is, a begin iterator that points to one container and an end iterator that points to a different container.**

Algorithm `for_each`

20 This example uses the `for_each` algorithm from **C++20's `std::ranges` namespace** twice. The first call in line 17 multiplies each `values` element by 2 and displays the result.

```
15    // output each element multiplied by two
16    std::cout << "\nDisplay each element multiplied by two: ";
17    std::ranges::for_each(values, [](auto i) {std::cout << i * 2 << " ";});
18
```

```
Display each element multiplied by two: 2 4 6 8
```

The `std::ranges::for_each` algorithm's two arguments are

- Concepts an `input_range` (`values`) containing the elements to process and
- a function with one argument, which is allowed to modify its argument in the `input_range`.

The `for_each` algorithm repeatedly calls the function in its second argument, passing one element at a time from its `input_range` argument.

Lambda with an Empty Introducer

Line 17's lambda multiplies its parameter `i` by 2 and displays the result. Lambdas begin with the **lambda introducer (`[]`)**, followed by a parameter list and a body. **This lambda introducer is empty, so it does not capture any local variables**. The parameter's type is `auto`, so this is a generic

lambda for which the compiler infers the type based on the context. Since we're iterating over `int` values, the compiler infers parameter `i`'s type as `int`.

The line 17 lambda, when specialized with `int`, is similar to the stand-alone function

```
void timesTwo(int i) {
   cout << i * 2 << " ";
}
```

Had we defined this function, we could have passed it to `for_each`, as in

```
std::ranges::for_each(values, timesTwo);
```

Lambda with a Nonempty Introducer: Capturing Local Variables

CG Line 21 calls `for_each` to total the elements of `values`. The **lambda introducer `[&sum]` captures the local variable `sum`** (defined in line 20) **by reference** (`&`), so the lambda can modify `sum`'s value.[12,13,14] The `for_each` algorithm passes each element of `values` to the lambda, which adds the element to the `sum`. Line 22 displays the `sum`. **Without the ampersand, `sum` would be captured by value and a compilation error would occur because a lambda's argument is treated as `const` by default.**

12. C++ Core Guidelines, "F.50: Use a Lambda When a Function Won't Do (to Capture Local Variables, or to Write a Local Function)." Accessed January 29, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-capture-vs-overload.
13. C++ Core Guidelines, "F.52: Prefer Capturing By Reference in Lambdas That Will Be Used Locally, Including Passed to Algorithms." Accessed January 29, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-reference-capture.
14. C++ Core Guidelines, "F.53: Avoid Capturing By Reference in Lambdas That Will Be Used Non-Locally, Including Returned, Stored on the Heap, or Passed to Another Thread." Accessed January 29, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-value-capture.

```
19    // add each element to sum
20    int sum{0}; // initialize sum to zero
21    std::ranges::for_each(values, [&sum](auto i) {sum += i;});
22    std::cout << "\nSum of value's elements is: " << sum << "\n";
23 }
Sum of value's elements is: 10
```

Lambda Introducers `[&]` and `[=]`

A **lambda introducer** can capture multiple variables from the enclosing function's scope by providing a comma-separated list of variables. You also may capture multiple variables using **lambda introducers** of the form `[&]` or `[=]`:

- The **lambda introducer `[&]`** indicates that every variable from the enclosing scope used in the **lambda's body** should be captured **by reference**.
- The **lambda introducer `[=]`** indicates that every variable from the enclosing scope used in the **lambda's body** should be captured **by value**.

SE It's better to specify which variables to capture. If you specify a list of variables to capture, each one preceded by `&` will be captured by reference.

Lambda Return Types

11 The compiler can infer a **lambda's return type** from the return statement in its body. You also can specify a lambda's return type explicitly using C++11's **trailing return type** syntax:

```
[](parameterList) -> type {lambdaBody}
```

The trailing return type (`-> type`) is placed between the parameter list's closing right parenthesis and the lambda's body.

14.4 Algorithms

20 Sections 14.4.1–14.4.14 demonstrate many of the standard library algorithms. Most of the algorithms we present are the C++20 `std::ranges` versions.

14.4.1 `fill`, `fill_n`, `generate` and `generate_n`

20 Figure 14.2 demonstrates the **C++20's `std::ranges` namespace's `fill`, `fill_n`, `generate` and `generate_n`** algorithms:

- Algorithms `fill` and `fill_n` set every element or the first *n* elements in a range to a specific value.
- Algorithms `generate` and `generate_n` use a **generator function**[15] to create values for every element or the first *n* elements in a range. The generator function takes no arguments and returns a value.

15. Not to be confused with a C++20 generator coroutine (Chapter 18, C++20 Coroutines).

Lines 9–12 define a `nextLetter` function, which generates letters. We'll also implement this as a lambda so you can see the similarities.[16] Line 15 defines `chars`—a 10-element `std::array` of `char` values that we'll manipulate in this example.

16. Function `nextLetter` is not thread-safe. Chapter 17 discusses creating thread-safe functions.

```
1  // fig14_02.cpp
2  // Algorithms fill, fill_n, generate and generate_n.
3  #include <algorithm> // algorithm definitions
4  #include <array> // array class-template definition
5  #include <iostream>
6  #include <iterator> // ostream_iterator
7
8  // returns the next letter (starts with A)
9  char nextLetter() {
10    static char letter{'A'};
11    return letter++;
12 }
13
14 int main() {
15    std::array<char, 10> chars{};
```

Fig. 14.2 Algorithms `fill`, `fill_n`, `generate` and `generate_n`.

`fill` Algorithm

20 Concepts Line 16 uses the **C++20 std::ranges:: fill algorithm** to place `'5'` in every `chars` element. The first argument must be an **`output_range`** so that the algorithm can

- iterate from the beginning to the end of the **range** by incrementing its **iterator** with `++` and
- **dereference the iterator** to write a new value into the current element.

Concepts An `array` has **`contiguous_iterator`**s, which support all iterator operations. So, **`fill`** can increment the iterators with `++`. Also, the array `chars` is non-`const`, so **`fill`** can dereference the iterators to write values into the range. Line 20 displays `chars`' elements. In this example's outputs, we use bold text to highlight each algorithm's changes to `chars`.

```
16    std::ranges::fill(chars, '5'); // fill chars with 5s
17
18    std::cout << "chars after filling with 5s: ";
19    std::ostream_iterator<char> output{std::cout, " "};
20    std::ranges::copy(chars, output);
21
chars after filling with 5s: 5 5 5 5 5 5 5 5 5 5
```

`fill_n` Algorithm

20 Concepts Line 23 uses the **C++20 std::ranges::fill_n algorithm** to place the character `'A'` (the third argument) in `chars`' first five elements (specified by the second argument). The first argument must be at least an **`output_iterator`**. An `array` object supports **`contiguous_iterator`**s, and `chars` is non-`const`, so calling `chars.begin()` returns an iterator that `fill_n` can use to write values into `chars`.

```
22    // fill first five elements of chars with 'A's
23    std::ranges::fill_n(chars.begin(), 5, 'A');
24
25    std::cout << "\nchars after filling five elements with 'A's: ";
26    std::ranges::copy(chars, output);
27
chars after filling five elements with 'A's: A A A A A 5 5 5 5 5
```

`generate` Algorithm

20 Concepts Line 29 uses the **C++20 std::ranges::generate algorithm** to generate values for every `chars` element. The first argument must be an **`output_range`**. The second argument is a function that takes no arguments and returns a value. Function `nextLetter` (lines 9–12) defines a `static` local `char` variable `letter` and initializes it to `'A'`. Line 11 returns the current `letter` value, postincrementing it for use in the next call to the function.

```
28    // generate values for all elements of chars with nextLetter
29    std::ranges::generate(chars, nextLetter);
30
31    std::cout << "\nchars after generating letters A-J: ";
32    std::ranges::copy(chars, output);
```

```
33
chars after generating letters A-J: A B C D E F G H I J
```

`generate_n` Algorithm

20 Concepts Line 35 uses the **C++20 std::ranges::generate_n algorithm** to place the result of each call to `nextLetter` in the five elements of `chars` starting from `chars.begin()`. The first argument must be at least an **`input_or_output_iterator`** that is **`indirectly_writable`** so the algorithm can dereference the iterator to write into the target range.

```
34    // generate values for first five elements of chars with nextLetter
35    std::ranges::generate_n(chars.begin(), 5, nextLetter);
36
37    std::cout << "\nchars after generating K-O into elements 0-4: ";
38    std::ranges::copy(chars, output);
39
chars after generating K-O into elements 0-4: K L M N O F G H I J
```

Using the `generate_n` Algorithm with a Lambda

20 Lines 41–46 call the **C++20 `std::ranges::generate_n` algorithm**, passing a no-argument lambda (lines 42–45) that returns a generated letter. The compiler infers from the `return` statement that the lambda's return type is `char`.

```
40    // generate values for first three elements of chars with a lambda
41    std::ranges::generate_n(chars.begin(), 3,
42       []{ // lambda that takes no arguments
43          static char letter{'A'};
44          return letter++;
45       }
46    );
47
48    std::cout << "\nchars after generating A-C into elements 0-2: ";
49    std::ranges::copy(chars, output);
50    std::cout << "\n";
51 }
chars after generating A-C into elements 0-2: A B C N O F G H I J
```

14.4.2 `equal`, `mismatch` and `lexicographical_compare`

20 Figure 14.3 demonstrates comparing sequences of values for equality using algorithms **`equal`**, **`mismatch`** and **`lexicographical_compare`** from **C++20's `std::ranges` namespace**. Lines 12–14 create and initialize three `array`s, then lines 17–22 display their contents.

```
1  // fig14_03.cpp
2  // Algorithms equal, mismatch and lexicographical_compare.
3  #include <algorithm> // algorithm definitions
4  #include <array> // array class-template definition
```

```
 5  #include <fmt/format.h> // C++20: This will be #include <format>
 6  #include <iomanip>
 7  #include <iostream>
 8  #include <iterator> // ostream_iterator
 9  #include <string>
10
11  int main() {
12     std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13     std::array a2{a1}; // initializes a2 with copy of a1
14     std::array a3{1, 2, 3, 4, 1000, 6, 7, 8, 9, 10};
15     std::ostream_iterator<int> output{std::cout, " "};
16
17     std::cout << "a1 contains: ";
18     std::ranges::copy(a1, output);
19     std::cout << "\na2 contains: ";
20     std::ranges::copy(a2, output);
21     std::cout << "\na3 contains: ";
22     std::ranges::copy(a3, output);
23
```

```
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 1 2 3 4 5 6 7 8 9 10
a3 contains: 1 2 3 4 1000 6 7 8 9 10
```

Fig. 14.3 Algorithms `equal`, `mismatch` and `lexicographical_compare`.

`equal` Algorithm

20 Concepts Lines 26 and 30 use the **C++20 std::ranges::equal algorithm** to compare two **`input_ranges`** for equality. The algorithm returns `false` if the sequences are not the same length. Otherwise, it compares each range's corresponding elements with the `==` operator, returning `true` if they're all equal and `false` otherwise. Line 26 compares the elements in `a1` to the elements in `a2`. In this example, `a1` and `a2` are equal. Line 30 compares `a1` and `a3`, which are not equal.

```
24     // compare a1 and a2 for equality
25     std::cout << fmt::format("\n\na1 is equal to a2: {}\n",
26        std::ranges::equal(a1, a2));
27
28     // compare a1 and a3 for equality
29     std::cout << fmt::format("a1 is equal to a3: {}\n",
30        std::ranges::equal(a1, a3));
31
```

```
a1 is equal to a2: true
a1 is equal to a3: false
```

`equal` Algorithm with Binary Predicate Function

Many standard library algorithms that compare elements allow you to pass a function that customizes how elements should be compared. For example, you can pass to the **`equal` algorithm** a function that receives as arguments the two elements to compare and returns a `bool` value indicating whether they are equal. Because the function receives two arguments and returns a `bool`, it's referred to as a **binary predicate function**. This can be useful in ranges containing objects for which an `==` operator is not defined or objects containing pointers. For example, you can

compare `Employee` objects for age, ID number, or location rather than comparing entire objects. You can compare what pointers refer to rather than comparing the addresses stored in the pointers.

20 The **C++20 `std::ranges` algorithms** also support **projections**, which enable algorithms to process subsets of each object in a range. For example, to sort `Employee` objects by their salaries, you can use a projection that selects each `Employee`'s salary. While sorting the `Employee`s, they'd be compared only by their salaries to determine sort order, and the `Employee` objects would be arranged accordingly. We'll perform this sort in Section 14.6.

`mismatch` Algorithm

20 The **C++20 `std::ranges::mismatch` algorithm** (line 33) compares two **`input_range`s**. The algorithm returns a `std::ranges::mismatch_result`, which contains **iterators** named `in1` and `in2`, pointing to the mismatched elements in each range. If all the elements match, `in1` is equal to the first **range's sentinel**, and `in2` is equal to the second **range's sentinel**. Line 33 infers the variable `location`'s type with `auto`. Line 35 determines the index of the mismatch with the expression

```
location.in1 - a1.begin()
```

which evaluates to the number of elements between the iterators—this is analogous to pointer arithmetic (Chapter 7). Like `equal`, `mismatch` can receive a **binary predicate function** to customize the comparisons (Section 14.6).

```
32    // check for mismatch between a1 and a3
33    auto location{std::ranges::mismatch(a1, a3)};
34    std::cout << fmt::format("a1 and a3 mismatch at index {} ({} vs. {})\n",
35       (location.in1 - a1.begin()), *location.in1, *location.in2);
36
a1 and a3 mismatch at index 4 (5 vs. 1000)
```

A Note Regarding `auto` and Algorithm Return Types

Template type declarations can become complex and error-prone quickly, as is commonly the case for standard library algorithm return types. When we initialize a variable with an algorithm's return value (as in line 33), we'll use **`auto`** to infer the type. To help you understand why, consider the **`std::ranges::mismatch` algorithm**'s return type in line 33:

```
std::ranges::mismatch_result<borrowed_iterator_t<R1>,
   borrowed_iterator_t<R2>>
```

`R1` and `R2` are the types of the ranges passed to `mismatch`. Based on the declarations of `a1` and `a3`, the algorithm's return type in line 33 is

```
std::ranges::mismatch_result<borrowed_iterator_t<array<int, 10>>,
   borrowed_iterator_t<array<int, 10>>>
```

17 As you can see, it's much more convenient to simply say **`auto`** and let the compiler determine this complex declaration for you. C++17 class-template argument deduction (CTAD) also could be used to infer the type arguments. So, we could declare the variable's type simply as `std::ranges::mismatch_result`.

lexicographical_compare Algorithm

20 Concepts The **C++20 std::ranges::lexicographical_compare algorithm** (lines 41–42) compares the contents of two **input_ranges**—in this case, strings. Like containers, **strings have iterators that enable them to be treated as C++20 ranges**. While iterating through the ranges, if there is a mismatch between their corresponding elements and the element in the first range is less than the corresponding element in the second range, the algorithm returns true. Otherwise, the algorithm returns false. This algorithm can be used to arrange sequences lexicographically. It also can receive a **binary predicate function** that returns true if its first argument should be considered less than its second. Of course, strings are comparable with the relational and equality operators, so line 42's call to function lexicographical_compare could be replaced with s1 < s2. This algorithm is primarily intended for use with ranges that cannot be compared directly with one another, such as two std::spans.

```
37    std::string s1{"HELLO"};
38    std::string s2{"BYE BYE"};
39
40    // perform lexicographical comparison of c1 and c2
41    std::cout << fmt::format("\"{}\" < \"{}\": {}\n", s1, s2,
42       std::ranges::lexicographical_compare(s1, s2));
43 }
```

```
"HELLO" < "BYE BYE": false
```

14.4.3 remove, remove_if, remove_copy and remove_copy_if

20 Figure 14.4 demonstrates removing values from a sequence with algorithms **remove**, **remove_if**, **remove_copy** and **remove_copy_if** from **C++20's std::ranges namespace**.[17] Line 9 creates a vector<int> that we'll use to initialize other vectors in this example.

17. As of January 2022, std::ranges::remove did not compile on the most recent clang++ compiler.

```
1  // fig14_04.cpp
2  // Algorithms remove, remove_if, remove_copy and remove_copy_if.
3  #include <algorithm> // algorithm definitions
4  #include <iostream>
5  #include <iterator> // ostream_iterator
6  #include <vector>
7
8  int main() {
9     std::vector init{10, 2, 15, 4, 10, 6};
10    std::ostream_iterator<int> output{std::cout, " "};
11
```

Fig. 14.4 Algorithms remove, remove_if, remove_copy and remove_copy_if.

remove Algorithm

20 Concepts Concepts Lines 12–14 initialize vector v1 with a copy of init's elements, then output v1's contents. Line 17 uses the **C++20 std::ranges::remove algorithm** to eliminate from v1 all elements with the value 10. The first argument must be a **forward_range**, which supports **forward_iterators**. The iterators also must be **std::permutable**, enabling operations such as

swapping and moving elements as this algorithm removes elements. A `vector` supports more powerful `random_access_iterator`s, so it can work with any algorithm that requires lesser iterators. **This algorithm does not destroy the eliminated elements.** Instead, it places the remaining elements at the beginning of the container and returns a subrange specifying the elements that are no longer valid. **That subrange should not be used, so its elements typically are erased from the container**, as we'll discuss momentarily.

```
12    std::vector v1{init}; // initialize with copy of init
13    std::cout << "v1: ";
14    std::ranges::copy(v1, output);
15
16    // remove all 10s from v1
17    auto removed{std::ranges::remove(v1, 10)};
18    v1.erase(removed.begin(), removed.end());
19    std::cout << "\nv1 after removing 10s: ";
20    std::ranges::copy(v1, output);
21
```

```
v1: 10 2 15 4 10 6
v1 after removing 10s: 2 15 4 6
```

Erase–Remove Idiom

Perf Line 18 uses the `vector`'s `erase` member function to delete the subrange `removed`, representing the invalid `vector` elements. This reduces the `vector`'s size to its remaining number of elements. Line 20 outputs the updated contents of `v1`.

The `remove` and `erase` combination in lines 17 and 18 performs a technique known as the **erase–remove idiom**.[18] You remove elements via `remove` or `remove_if`, then call the `vector`'s `erase` member function to eliminate the now unused elements.

18. "Erase–remove idiom." Wikipedia. Wikimedia Foundation. Accessed January 29, 2022. https://en.wikipedia.org/wiki/Erase-remove_idiom.

The **common-range** `std::remove` and `std::remove_if` **algorithms** each return a single **iterator** pointing to the first invalid element in the `vector`. When using the **common-range algorithms**, you can perform the **erase–remove idiom** in one statement, as in

```
v1.erase(std::remove(v1.begin(), v1.end(), 10), v1.end());
```

Unfortunately, container member functions like `erase` do not yet support C++20 ranges.

20 C++20 `std::erase` and `std::erase_if`

Perf To simplify the **erase–remove idiom**, C++20 added the `std::erase` and `std::erase_if` functions. Both are overloaded in the headers `<string>`, `<vector>`, `<deque>`, `<list>` and `<forward_list>`, and `std::erase_if` is overloaded in `<map>`, `<set>`, `<unordered_map>` and `<unordered_set>`. Each function performs the erase–remove idiom in a single function call that receives a given container as its first argument. For example, lines 17–18 can be replaced with

```
std::erase(v1, 10);
```

`remove_copy` Algorithm

20 Concepts Line 28 uses the **C++20** **`std::ranges::remove_copy`** **algorithm** to copy all of `v2`'s elements that do not have the value `10` into the `vector` `c1`. The first argument must be an **`input_range`**, which supports **`input_iterator`**s for reading from a range. Again, a `vector` supports more powerful **`random_access_iterator`**s, so it meets **`remove_copy`**'s minimum requirements. We'll discuss the second argument in a moment. Line 30 outputs all of `c1`'s elements.

```
22    std::vector v2{init}; // initialize with copy of init
23    std::cout << "\n\nv2: ";
24    std::ranges::copy(v2, output);
25
26    // copy from v2 to c1, removing 10s in the process
27    std::vector<int> c1{};
28    std::ranges::remove_copy(v2, std::back_inserter(c1), 10);
29    std::cout << "\nc1 after copying v2 without 10s: ";
30    std::ranges::copy(c1, output);
31
```

```
v2: 10 2 15 4 10 6
c1 after copying v2 without 10s: 2 15 4 6
```

Iterator Adapters: `back_inserter`, `front_inserter` and `inserter`

Err The second argument specifies an **output iterator** indicating where the copied elements will be written—typically another container. The **`remove_copy`** **algorithm** does not check whether the target container has enough room to store all the copied elements. So the iterator in the second argument must refer to a container with enough room for all the elements. Rather than preallocating memory in advance, you can use an **iterator adaptor** to **grow a container**, allowing it to allocate more elements as you insert new ones. In line 28, the expression `std::back_inserter(c1)` uses the **`back_inserter`** **iterator adaptor** (header **`<iterator>`**) create a **`back_insert_iterator`**. When the algorithm uses this iterator to insert an element in `c1`, the iterator calls the container's **`push_back`** **function** to place the element at the end of `c1`. If the container needs more space, it grows to accommodate the new element. **A `back_insert_iterator` cannot be used with `array`s because they are fixed-size containers and do not have a `push_back` function.**

There are two other **inserters**:

- A **`front_inserter`** creates a **`front_insert_iterator`** that uses a container's **`push_front`** member function to insert an element at the beginning of a container.
- An **`inserter`** creates a **`insert_iterator`** that uses the container's **`insert`** member function to insert an element in the container specified by the adaptor's first argument at the location specified by the **iterator** in the adaptor's second argument.

`remove_if` Algorithm

Concepts Line 38 calls the **C++20** **`std::ranges::remove_if`** **algorithm** to delete from `v3` all elements for which the **unary predicate function** **`greaterThan9`** returns `true`. The first argument must be a **`forward_range`** that enables **`remove_if`** to read the elements in the range. A **unary predicate function** must receive one parameter and return a `bool` value.

```
32    std::vector v3{init}; // initialize with copy of init
33    std::cout << "\n\nv3: ";
34    std::ranges::copy(v3, output);
35
36    // remove elements greater than 9 from v3
37    auto greaterThan9{[](auto x) {return x > 9;}};
38    auto removed2{std::ranges::remove_if(v3, greaterThan9)};
39    v3.erase(removed2.begin(), removed2.end());
40    std::cout << "\nv3 after removing elements greater than 9: ";
41    std::ranges::copy(v3, output);
42
```

```
v3: 10 2 15 4 10 6
v3 after removing elements greater than 9: 2 4 6
```

Line 37 defines a **unary predicate function** as the generic lambda

```
[](auto x){return x > 9;}
```

which returns `true` if its argument is greater than 9; otherwise, it returns `false`. The compiler uses `auto` type inference for both the lambda's parameter and return types:

- The `vector` contains `int`s, so the compiler infers the parameter's type as `int`.
- The lambda returns the result of evaluating a condition, so the compiler infers the return type as `bool`.

20 Like **`remove`**, **`remove_if`** does not modify the number of elements in the container. Instead, it places at the beginning of the container all elements that are not eliminated. It returns a subrange of elements that are no longer valid. We use that subrange in line 39 to erase those elements in `v3`. Line 41 outputs the updated contents of `v3`. Lines 38–39 can be replaced with C++20's **`std::erase_if`**:

```
std::erase_if(v3, greaterThan9);
```

`remove_copy_if` Algorithm

20 Concepts Line 49 calls the **C++20 `std::ranges::remove_copy_if` algorithm** to copy the elements from its **`input_range`** argument `v4` for which a **unary predicate function** (the **lambda** at line 37) returns `true`. The second argument must be an **output iterator** so the element being copied can be written into the destination range. We use a **`back_inserter`** to insert the copied elements into the `vector` `c2`. Line 51 outputs the contents of `c2`.

```
43    std::vector v4{init}; // initialize with copy of init
44    std::cout << "\n\nv4: ";
45    std::ranges::copy(v4, output);
46
47    // copy elements from v4 to c2, removing elements greater than 9
48    std::vector<int> c2{};
49    std::ranges::remove_copy_if(v4, std::back_inserter(c2), greaterThan9);
50    std::cout << "\nc2 after copying v4 without elements greater than 9: ";
51    std::ranges::copy(c2, output);
52    std::cout << "\n";
53  }
```

```
v4: 10 2 15 4 10 6
```

```
c2 after copying v4 without elements greater than 9: 2 4 6
```

14.4.4 `replace`, `replace_if`, `replace_copy` and `replace_copy_if`

20 Figure 14.5 demonstrates replacing values in a sequence using **C++20's** **`std::ranges`** algorithms **`replace`**, **`replace_if`**, **`replace_copy`** and **`replace_copy_if`**.

```
1  // fig14_05.cpp
2  // Algorithms replace, replace_if, replace_copy and replace_copy_if.
3  #include <algorithm>
4  #include <array>
5  #include <iostream>
6  #include <iterator> // ostream_iterator
7
8  int main() {
9    std::ostream_iterator<int> output{std::cout, " "};
10
```

Fig. 14.5 Algorithms `replace`, `replace_if`, `replace_copy` and `replace_copy_if`.

`replace` Algorithm

20 Concepts Line 16 calls the **C++20** **`std::ranges::replace`** algorithm to replace all `10`s in the **`input_range`** `a1` with `100`s. The first argument must support **`indirectly_writable`** **iterators**, so **`replace`** can **dereference the iterators** to replace values in the **range**. The `array` `a1` is non-`const`, so its **iterators** can be used to write into the `array`.

```
11    std::array a1{10, 2, 15, 4, 10, 6};
12    std::cout << "a1: ";
13    std::ranges::copy(a1, output);
14
15    // replace all 10s in a1 with 100
16    std::ranges::replace(a1, 10, 100);
17    std::cout << "\na1 after replacing 10s with 100s: ";
18    std::ranges::copy(a1, output);
19
```

```
a1: 10 2 15 4 10 6
a1 after replacing 10s with 100s: 100 2 15 4 100 6
```

`replace_copy` Algorithm

20 Concepts Line 26 calls the **C++20** **`std::ranges::replace_copy`** algorithm to copy all elements in the **`input_range`** `a2`, replacing each `10` with `100`. The second argument must be an **output iterator** representing where to write the copied elements. The algorithm copies or replaces every element in the **`input_range`**, so we allocated `c1` with the same number of elements as `a2`, but we could have used an empty `vector` and a **`back_inserter`**.

```
20    std::array a2{10, 2, 15, 4, 10, 6};
21    std::array<int, a2.size()> c1{};
```

```
22    std::cout << "\n\na2: ";
23    std::ranges::copy(a2, output);
24
25    // copy from a2 to c1, replacing 10s with 100s
26    std::ranges::replace_copy(a2, c1.begin(), 10, 100);
27    std::cout << "\nc1 after replacing a2's 10s with 100s: ";
28    std::ranges::copy(c1, output);
29
```

```
a2: 10 2 15 4 10 6
c1 after replacing a2's 10s with 100s: 100 2 15 4 100 6
```

`replace_if` Algorithm

20 Concepts Line 36 calls the **C++20** **`std::ranges::replace_if` algorithm** to replace each element in the **`input_range`** `a3` for which a **unary predicate function** (**`greaterThan9`** in line 35) returns `true`. The first argument must support **output iterators**, so `replace_if` can replace values in the range. Here, we replace each value greater than 9 with 100.

```
30    std::array a3{10, 2, 15, 4, 10, 6};
31    std::cout << "\n\na3: ";
32    std::ranges::copy(a3, output);
33
34    // replace values greater than 9 in a3 with 100
35    auto greaterThan9{[](auto x) {return x > 9;}};
36    std::ranges::replace_if(a3, greaterThan9, 100);
37    std::cout << "\na3 after replacing values greater than 9 with 100s: ";
38    std::ranges::copy(a3, output);
39
```

```
a3: 10 2 15 4 10 6
a3 after replacing values greater than 9 with 100s: 100 2 100 4 100 6
```

`replace_copy_if` Algorithm

20 Concepts Line 46 calls the **C++20** **`std::ranges::replace_copy_if` algorithm** to copy all elements in the **`input_range`** `a4` and if the **unary predicate function** returns `true`, replace their values. Here, we replace values greater than 9 with the value `100`. The copied or replaced elements are placed in `c2`, starting at position `c2.begin()`, which must be an **output iterator**.

```
40    std::array a4{10, 2, 15, 4, 10, 6};
41    std::array<int, a4.size()> c2{};
42    std::cout << "\n\na4: ";
43    std::ranges::copy(a4, output);
44
45    // copy a4 to c2, replacing elements greater than 9 with 100
46    std::ranges::replace_copy_if(a4, c2.begin(), greaterThan9, 100);
47    std::cout << "\nc2 after replacing a4's values "
48       << "greater than 9 with 100s: ";
49    std::ranges::copy(c2, output);
50    std::cout << "\n";
51 }
```

```
a4: 10 2 15 4 10 6
c2 after replacing a4's values greater than 9 with 100s: 100 2 100 4 100 6
```

14.4.5 Shuffling, Counting, and Minimum and Maximum Element Algorithms

20 Figure 14.6 demonstrates `shuffle`, `count`, `count_if`, `min_element`, `max_element` and `minmax_element` from **C++20's** `std::ranges` **namespace**. We also use the `transform` algorithm to cube values in a range.

```
1  // fig14_06.cpp
2  // Shuffling, counting, and minimum and maximum element algorithms.
3  #include <algorithm>
4  #include <array>
5  #include <iostream>
6  #include <iterator>
7  #include <random>
8
9  int main() {
10    std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
11    std::ostream_iterator<int> output{std::cout, " "};
12
13    std::cout << "a1: ";
14    std::ranges::copy(a1, output);
15
```

```
a1: 1 2 3 4 5 6 7 8 9 10
```

Fig. 14.6 Mathematical algorithms of the standard library.

`shuffle` Algorithm

20 Concepts Line 18 calls the **C++20** `std::ranges::shuffle` **algorithm** to randomly reorder `a1`'s elements. This requires a `random_access_range` with `random_access_iterator`s and thus can be used with the containers `array`, `vector` and `deque` (and with built-in arrays). The algorithm's declaration indicates that the **range**'s **iterators** must be `permutable`, which permits operations such as swapping and moving elements. A non-`const array`'s iterators allow such operations. The `shuffle` algorithm's second argument is a **C++11 random-number-generator engine** (Section 5.8). Line 20 displays the shuffled results.

```
16    // create random-number engine and use it to help shuffle a1
17    std::default_random_engine randomEngine{std::random_device{}()};
18    std::ranges::shuffle(a1, randomEngine); // randomly order elements
19    std::cout << "\na1 shuffled: ";
20    std::ranges::copy(a1, output);
21
```

```
a1 shuffled: 5 4 7 6 3 9 1 8 10 2
```

`count` Algorithm

20 Concepts Line 27 calls the **C++20** **`std::ranges::count`** **algorithm** to count the elements with a specific value (in this case, `8`) in `a2`. The first argument must be an **`input_range`**, so **count** can read elements in the range.

```
22    std::array a2{100, 2, 8, 1, 50, 3, 8, 8, 9, 10};
23    std::cout << "\n\na2: ";
24    std::ranges::copy(a2, output);
25
26    // count number of elements in a2 with value 8
27    auto result1{std::ranges::count(a2, 8)};
28    std::cout << "\nCount of 8s in a2: " << result1;
29
```

```
a2: 100 2 8 1 50 3 8 8 9 10
Count of 8s in a2: 3
```

`count_if` Algorithm

20 Concepts Line 31 calls the **C++20** **`std::ranges::count_if`** **algorithm** to count in its **`input_range`** argument `a2` elements for which a **unary predicate** function returns `true`. Once again, we use a **lambda** to define a **unary predicate** that returns `true` for a value greater than 9.

```
30    // count number of elements in a2 that are greater than 9
31    auto result2{std::ranges::count_if(a2, [](auto x){return x > 9;})};
32    std::cout << "\nCount of a2 elements greater than 9: " << result2;
33
```

```
Count of a2 elements greater than 9: 3
```

`min_element` Algorithm

20 Concepts Err Line 35 calls the **C++20** **`std::ranges::min_element`** **algorithm** to locate the smallest element in its **`forward_range`** argument `a2`. Such a **range** supports **`forward_iterators`**, which allow **`min_element`** to read elements from the **range**. The algorithm returns an **iterator** located at the first occurrence of the **range's smallest element** or the **range's sentinel if the range is empty**. As with many algorithms that compare elements, you can provide a custom **binary predicate function** that specifies how to compare the elements and returns `true` if the first argument should be considered less than the second. **Before dereferencing an iterator that might represent a range's sentinel, you should check that it does not match the sentinel**, as in line 35.

```
34    // locate minimum element in a2
35    if (auto result{std::ranges::min_element(a2)}; result != a2.end()) {
36       std::cout << "\n\na2 minimum element: " << *result;
37    }
38
```

```
a2 minimum element: 1
```

`max_element` Algorithm

20 Concepts Line 40 calls the **C++20** **`std::ranges::max_element`** **algorithm** to locate the largest element in its **`forward_range`** argument `a2`. Such a **range** supports **`forward_iterators`**, which allow **`max_element`** to read elements from the **range**. The algorithm returns an **iterator** located at the first occurrence of the **range's largest element** or the **range's sentinel if the range is empty**. Again, you can provide a custom **binary predicate function** specifying how to compare the elements.

```
39    // locate maximum element in a2
40    if (auto result{std::ranges::max_element(a2)}; result != a2.end()) {
41       std::cout << "\na2 maximum element: " << *result;
42    }
43
```

```
a2 maximum element: 100
```

`minmax_element` Algorithm

20 Concepts Line 45 calls the **C++20** **`std::ranges::minmax_element`** **algorithm** to locate the smallest and largest elements in its **`forward_range`** argument `a2`. The algorithm returns a **`std::ranges::minmax_element_result`** containing iterators named **`min`** and **`max`** aimed at the smallest and largest elements, respectively:

- If there are duplicate smallest elements, the first **iterator** is located at the **first of the smallest values**.
- If there are duplicate largest elements, **the second iterator is located at the last of the largest values**—note that this is different from how `max_element` works.

Again, you can provide a custom **binary predicate function** specifying how to compare the elements.

```
44    // locate minimum and maximum elements in a2
45    auto [min, max]{std::ranges::minmax_element(a2)};
46    std::cout << "\na2 minimum and maximum elements: "
47       << *min << " and " << *max;
48
```

```
a2 minimum and maximum elements: 1 and 100
```

C++17 Structured Bindings

17 Line 45 uses the **`minmax_element_result`** returned by **`minmax_element`** to initialize the variables `min` and `max` using a C++17 **structured binding declaration**.[19,20] This is sometimes referred to as **unpacking the elements** and can be used to extract into individual variables the elements of a built-in array, `std::array` object, `std::pair` or `std::tuple` (each of which has a size that's known at compile-time). **Structured bindings** also can be used to unpack the `public` data members of a `struct` or class `object`. In each Chapter 13 example in which a function returned a `std::pair`, we also could have used structured bindings to unpack its members.

19. "Structured Binding Declaration." Accessed January 29, 2022. https://en.cppreference.com/w/cpp/language/structured_binding.
20. Dominik Berner, "Quick and Easy Unpacking in C++ with Structured Bindings." May 24, 2018. Accessed January 29, 2022. https://dominikberner.ch/structured-bindings/.

`transform` Algorithm

20 Concepts © Concepts © Lines 51–52 use the **C++20** **`std::ranges::transform`** **algorithm** to transform the elements of its **`input_range`** **`a1`** to new values. Each new value is written into the range specified by the algorithm's second argument, which must be an **output iterator** and can point to the same container as the **`input_range`** or a different container. The function provided as **`transform`**'s third argument receives a value and returns a new value. There is no guarantee of the order in which this function will be called for the range's elements, so it should not modify the **`input_range`**'s elements. To modify the original range's elements, use the **`std::foreach`** algorithm.[21]

21. "Algorithms Library—mutating Sequence Operations—Transform." Accessed January 29, 2022. https://timsong-cpp.github.io/cppwp/n4861/alg.transform.

```
49    // calculate cube of each element in a1; place results in cubes
50    std::array<int, a1.size()> cubes{};
51    std::ranges::transform(a1, cubes.begin(),
52       [](auto x){return x * x * x;});
53    std::cout << "\n\na1 values cubed: ";
54    std::ranges::copy(cubes, output);
55    std::cout << "\n";
56 }
```

```
a1 values cubed: 125 64 343 216 27 729 1 512 1000 8
```

Concepts © A **`transform`** overload accepts two **`input_range`**s, an **output iterator** and a function that takes two arguments and returns a result. This version of **`transform`** passes corresponding elements from each **`input_range`** to its function argument, then outputs the function's result via the **output iterator**.

14.4.6 Searching and Sorting Algorithms

20 Figure 14.7 demonstrates some basic searching and sorting algorithms, including **`find`**, **`find_if`**, **`sort`**, **`binary_search`**, **`all_of`**, **`any_of`**, **`none_of`** and **`find_if_not`** from **C++20's** **`std::ranges`** **namespace**.

```
1  // fig14_07.cpp
2  // Standard library search and sort algorithms.
3  #include <algorithm> // algorithm definitions
4  #include <array> // array class-template definition
5  #include <iostream>
6  #include <iterator>
7
8  int main() {
9     std::array values{10, 2, 17, 5, 16, 8, 13, 11, 20, 7};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output); // display output vector
14
```

```
values contains: 10 2 17 5 16 8 13 11 20 7
```

Fig. 14.7 Standard library search and sort algorithms.

`find` Algorithm

20 Concepts The **C++20 `std::ranges::find` algorithm** (line 16) **performs an O(n) linear search** to find a value (`16`) in its **`input_range`** argument (`values`). The algorithm returns an iterator positioned at the first element containing the value; otherwise, it returns the range's sentinel (demonstrated by lines 24–29). We use the returned iterator in line 17 to calculate the index position at which the value was found.

```
15    // locate first occurrence of 16 in values
16    if (auto loc1{std::ranges::find(values, 16)}; loc1 != values.cend()) {
17       std::cout << "\n\nFound 16 at index: " << (loc1 - values.cbegin());
18    }
19    else { // 16 not found
20       std::cout << "\n\n16 not found";
21    }
22
23    // locate first occurrence of 100 in values
24    if (auto loc2{std::ranges::find(values, 100)}; loc2 != values.cend()) {
25       std::cout << "\nFound 100 at index: " << (loc2 - values.cbegin());
26    }
27    else { // 100 not found
28       std::cout << "\n100 not found";
29    }
30
```

```
Found 16 at index: 4
100 not found
```

`find_if` Algorithm

20 Concepts The **C++20 `std::ranges::find_if` algorithm** (line 35) **performs an O(n) linear search** to locate the first value in its **`input_range`** argument (`values`) for which a **unary predicate function** (**`isGreaterThan10`** from line 32) returns `true`. The algorithm returns an iterator positioned at the first element containing a value for which the **predicate function** returns `true`; otherwise, it returns the range's sentinel.

```
31    // create variable to store lambda for reuse later
32    auto isGreaterThan10{[](auto x){return x > 10;}};
33
34    // locate first occurrence of value greater than 10 in values
35    auto loc3{std::ranges::find_if(values, isGreaterThan10)};
36
37    if (loc3 != values.cend()) { // found value greater than 10
38       std::cout << "\n\nFirst value greater than 10: " << *loc3
39          << "\nfound at index: " << (loc3 - values.cbegin());
40    }
41    else { // value greater than 10 not found
42       std::cout << "\n\nNo values greater than 10 were found";
43    }
44
```

```
First value greater than 10: 17
```

```
found at index: 2
```

`sort` Algorithm

20 Concepts The **C++20** **`std::ranges::sort`** **algorithm** (line 46) **performs an O(n log n) sort** that arranges the elements in its argument `values` into ascending order. The argument must be a **`random_access_range`**, which supports **`random_access_iterator`**s and thus can be used with the containers **`array`**, **`vector`** and **`deque`** (and with built-in arrays). This algorithm also can receive a **binary predicate function** taking two arguments and returning a `bool` indicating the **sorting order**. The predicate compares two values from the sequence being sorted. If the binary predicate returns `true`, the two elements are already in sorted order; otherwise, the two elements need to be reordered in the sequence.

```
45    // sort elements of values
46    std::ranges::sort(values);
47    std::cout << "\n\nvalues after sort: ";
48    std::ranges::copy(values, output);
49
```

```
values after sort: 2 5 7 8 10 11 13 16 17 20
```

`binary_search` Algorithm

20 Concepts SE The **C++20** **`std::ranges::binary_search`** **algorithm** (line 51) **performs an O(log n) binary search** to determine whether a value (`13`) is in its **`forward_range`** argument (`values`). The **range** must be **sorted in ascending order**. The algorithm returns a `bool` indicating whether the value was found in the sequence. Line 59 demonstrates a call to `binary_search` for which the value is not found. This algorithm also can receive a **binary predicate function** that takes two arguments and returns a `bool`. The function should return `true` if the two elements being compared are in sorted order. **If you need to know the search key's location in the container, use the `lower_bound` or `find` algorithms rather than `binary_search`.**

```
50    // use binary_search to check whether 13 exists in values
51    if (std::ranges::binary_search(values, 13)) {
52       std::cout << "\n\n13 was found in values";
53    }
54    else {
55       std::cout << "\n\n13 was not found in values";
56    }
57
58    // use binary_search to check whether 100 exists in values
59    if (std::ranges::binary_search(values, 100)) {
60       std::cout << "\n100 was found in values";
61    }
62    else {
63       std::cout << "\n100 was not found in values";
64    }
65
```

```
13 was found in values
100 was not found in values
```

`all_of` Algorithm

20 Concepts The **C++20** **`std::ranges::all_of`** **algorithm** (line 67) **performs an O(n) linear search** to determine whether the **unary predicate function** in its second argument (in this case, the **lambda** **`isGreaterThan10`**) returns `true` for **all of the elements** in its **`input_range`** argument (`values`). If so, **`all_of`** returns `true`; otherwise, it returns `false`.

```
66    // determine whether all of values' elements are greater than 10
67    if (std::ranges::all_of(values, isGreaterThan10)) {
68       std::cout << "\n\nAll values elements are greater than 10";
69    }
70    else {
71       std::cout << "\n\nSome values elements are not greater than 10";
72    }
73
```

```
Some values elements are not greater than 10
```

`any_of` Algorithm

20 Concepts The **C++20** **`std::ranges::any_of`** **algorithm** (line 75) **performs an O(n) linear search** to determine whether the **unary predicate function** in its second argument (in this case, the **lambda** **`isGreaterThan10`**) returns `true` for **at least one element** in its **`input_range`** argument (`values`). If so, **`any_of`** returns `true`; otherwise, it returns `false`.

```
74    // determine whether any of values' elements are greater than 10
75    if (std::ranges::any_of(values, isGreaterThan10)) {
76       std::cout << "\n\nSome values elements are greater than 10";
77    }
78    else {
79       std::cout << "\n\nNo values elements are greater than 10";
80    }
81
```

```
Some values elements are greater than 10
```

`none_of` Algorithm

20 Concepts The **C++20** **`std::ranges::none_of`** **algorithm** (line 83) **performs an O(n) linear search** to determine whether the **unary predicate function** in its second argument (in this case, the **lambda** **`isGreaterThan10`**) returns `false` for **all of the elements** in its **`input_range`** argument (`values`). If so, **`none_of`** returns `true`; otherwise, it returns `false`.

```
82    // determine whether none of values' elements are greater than 10
83    if (std::ranges::none_of(values, isGreaterThan10)) {
84       std::cout << "\n\nNo values elements are greater than 10";
85    }
86    else {
87       std::cout << "\n\nSome values elements are greater than 10";
88    }
```

```
89
Some values elements are greater than 10
```

`find_if_not` Algorithm

20 Concepts The **C++20** **`std::ranges::find_if_not`** **algorithm** (line 91) **performs an O(n) linear search** to locate the first value in its **`input_range`** argument (`values`) for which a **unary predicate function** (the **lambda** **`isGreaterThan10`**) returns `false`. The algorithm returns an iterator that's positioned at the first element containing a value for which the **predicate function** returns `false`; otherwise, it returns the range's sentinel.

```
90    // locate first occurrence of value that is not greater than 10
91    auto loc4{std::ranges::find_if_not(values, isGreaterThan10)};
92
93    if (loc4 != values.cend()) { // found a value less than or equal to 10
94       std::cout << "\n\nFirst value not greater than 10: " << *loc4
95          << "\nfound at index: " << (loc4 - values.cbegin());
96    }
97    else { // no values less than or equal to 10 were found
98       std::cout << "\n\nOnly values greater than 10 were found";
99    }
100
101    std::cout << "\n";
102 }
First value not greater than 10: 2
found at index: 0
```

14.4.7 `swap`, `iter_swap` and `swap_ranges`

20 Figure 14.8 demonstrates algorithms for swapping elements—**`swap`** and **`iter_swap`** from the `std` namespace and algorithm **`swap_ranges`** from **C++20's** **`std::ranges`** **namespace**.

```
1  // fig14_08.cpp
2  // Algorithms swap, iter_swap and swap_ranges.
3  #include <algorithm>
4  #include <array>
5  #include <iostream>
6  #include <iterator>
7
8  int main() {
9     std::array values{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
10    std::ostream_iterator<int> output{std::cout, " "};
11
12    std::cout << "values contains: ";
13    std::ranges::copy(values, output);
14
values contains: 1 2 3 4 5 6 7 8 9 10
```

Fig. 14.8 Algorithms `swap`, `iter_swap` and `swap_ranges`.

`swap` Algorithm

Line 15 uses the `std::swap` **algorithm** to exchange its two arguments' values—this is not a range or common-range algorithm. The function receives references to the two values being exchanged. In this case, we pass references to the `array`'s first and second elements.

```
15    std::swap(values[0], values[1]); // swap elements at index 0 and 1
16
17    std::cout << "\nafter std::swap of values[0] and values[1]: ";
18    std::ranges::copy(values, output);
19
after std::swap of values[0] and values[1]: 2 1 3 4 5 6 7 8 9 10
```

`iter_swap` Algorithm

Line 21 uses the `std::iter_swap` **algorithm** to exchange the two elements specified by its **common-range forward iterator** arguments. The iterators can refer to any two elements of the same type.

```
20    // use iterators to swap elements at locations 0 and 1
21    std::iter_swap(values.begin(), values.begin() + 1);
22    std::cout << "\nafter std::iter_swap of values[0] and values[1]: ";
23    std::ranges::copy(values, output);
24
after std::iter_swap of values[0] and values[1]: 1 2 3 4 5 6 7 8 9 10
```

`swap_ranges` Algorithm

20 Concepts Concepts Line 31 uses the **C++20** `std::ranges::swap_ranges` **algorithm** to exchange the elements of its two `input_range` arguments. If the ranges are not the same length, the algorithm swaps the shorter sequence with the corresponding elements in the longer sequence. The **ranges** also must support `indirectly_swappable` **iterators**, so the algorithm can dereference the iterators to swap to the corresponding elements in each range.

```
25    // swap values and values2
26    std::array values2{10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
27    std::cout << "\n\nBefore swap_ranges\nvalues contains: ";
28    std::ranges::copy(values, output);
29    std::cout << "\nvalues2 contains: ";
30    std::ranges::copy(values2, output);
31    std::ranges::swap_ranges(values, values2);
32    std::cout << "\n\nAfter swap_ranges\nvalues contains: ";
33    std::ranges::copy(values, output);
34    std::cout << "\nvalues2 contains: ";
35    std::ranges::copy(values2, output);
36
Before swap_ranges
values contains: 1 2 3 4 5 6 7 8 9 10
values2 contains: 10 9 8 7 6 5 4 3 2 1

After swap_ranges
```

```
values contains: 10 9 8 7 6 5 4 3 2 1
values2 contains: 1 2 3 4 5 6 7 8 9 10
```

20 Concepts Lines 38–39 call the **C++20** `std::ranges::swap_ranges` overload that enables you to specify the portions of two `input_range`s to swap. Here, we swap the first five elements of `values` with the first five elements of `values2`, specifying the two ranges to swap as **iterator pairs**:

- `values.begin()` and `values.begin() + 5` indicate the first five elements in `values`.
- `values2.begin()` and `values2.begin() + 5` indicate the first five elements in `values2`.

Specifying iterator pairs also works with the **common ranges version** in the `std` namespace. In this example, the two **ranges** are in different containers, but **the ranges can be in the same container**, in which case **the ranges must not overlap**.

```
37    // swap first five elements of values and values2
38    std::ranges::swap_ranges(values.begin(), values.begin() + 5,
39       values2.begin(), values2.begin() + 5);
40
41    std::cout << "\n\nAfter swap_ranges for 5 elements"
42       << "\nvalues contains: ";
43    std::ranges::copy(values, output);
44    std::cout << "\nvalues2 contains: ";
45    std::ranges::copy(values2, output);
46    std::cout << "\n";
47 }
```

```
After swap_ranges for 5 elements
values contains: 1 2 3 4 5 5 4 3 2 1
values2 contains: 10 9 8 7 6 6 7 8 9 10
```

14.4.8 copy_backward, merge, unique, reverse, copy_if and copy_n

20 Figure 14.9 demonstrates algorithms **`copy_backward`**, **`merge`**, **`unique`**, **`reverse`**, **`copy_if`** and **`copy_n`** from **C++20's `std::ranges` namespace.**[22]

22. As of January 2022, `std::ranges::unique` did not compile on the most recent `clang++ compiler`.

```
1  // fig14_09.cpp
2  // Algorithms copy_backward, merge, unique, reverse, copy_if and copy_n.
3  #include <algorithm>
4  #include <array>
5  #include <iostream>
6  #include <iterator>
7  #include <vector>
8
9  int main() {
10    std::array a1{1, 3, 5, 7, 9};
11    std::array a2{2, 4, 5, 7, 9};
12    std::ostream_iterator<int> output{std::cout, " "};
13
14    std::cout << "array a1 contains: ";
15    std::ranges::copy(a1, output); // display a1
16    std::cout << "\narray a2 contains: ";
17    std::ranges::copy(a2, output); // display a2
```

```
18
array a1 contains: 1 3 5 7 9
array a2 contains: 2 4 5 7 9
```

Fig. 14.9 Algorithms `copy_backward`, `merge`, `unique`, `reverse`, `copy_if` and `copy_n`.

`copy_backward` Algorithm

20 Concepts The **C++20** **`std::ranges::copy_backward`** **algorithm** (line 21) copies its first argument's **`bidirectional_range`** (`a1`) into the range specified by the **output iterator** in its second argument—in this case, the end of the target container (`results.end()`). The algorithm copies the elements in reverse order, placing each into the target container **starting from the element before `results.end()` and working toward the beginning of the container**. The algorithm returns a **`std::ranges::copy_backward_result`**, containing two iterators:

- The first is positioned at `a1.end()`.
- The second is positioned at the last element copied into the target range—that is, the beginning of `results` because the copy is performed backward.

Though the elements are copied in reverse order, they're placed in `results` in the same order as `a1`. One difference between **`copy`** and **`copy_backward`** is that

- the **iterator** returned from **`copy`** is positioned **after the last element copied** and
- the one returned from **`copy_backward`** is positioned **at the last element copied**, which is the first element in the range.

```
19    // place elements of a1 into results in reverse order
20    std::array<int, a1.size()> results{};
21    std::ranges::copy_backward(a1, results.end());
22    std::cout << "\n\nAfter copy_backward, results contains: ";
23    std::ranges::copy(results, output);
24
After copy_backward, results contains: 1 3 5 7 9
```

`move` and `move_backward` Algorithm

20 You can use **move semantics** with ranges.

The **C++20** **`std::ranges`** **algorithms** **`move`** and **`move_backward`** (from header `<algorithm>`) work like the **`copy`** and **`copy_backward`** **algorithms**, but move the elements in the specified ranges rather than copying them.

`merge` Algorithm

20 The **C++20** **`std::ranges::merge`** **algorithm** (line 27) combines two **`input_range`**s that are each **sorted in ascending order** and writes the results to the target container specified by the third argument's **output iterator**. After this operation, `results2` contains both **ranges**' values in sorted order. A second version of **`merge`** takes **iterator/sentinel pairs** representing the two ranges. Both versions allow you to provide a **binary predicate function** that specifies the sorting order by comparing its two arguments and returning `true` if the first should be considered less than the second for ordering purposes.

```
25    // merge elements of a1 and a2 into results2 in sorted order
26    std::array<int, a1.size() + a2.size()> results2{};
```

```
27     std::ranges::merge(a1, a2, results2.begin());
28
29     std::cout << "\n\nAfter merge of a1 and a2, results2 contains: ";
30     std::ranges::copy(results2, output);
31
After merge of a1 and a2, results2 contains: 1 2 3 4 5 5 7 7 9 9
```

`unique` Algorithm

20 Concepts The **C++20** **`std::ranges::unique`** **algorithm** (line 34) determines the unique values in the sorted range of values specified by its **`forward_range`** argument. After **`unique`** is applied to a **sorted range** with duplicate values, a single copy of each value remains in the range. The algorithm returns a subrange from the element after the last unique value through the end of the original range. **The values of all elements in the container after the last unique value are undefined. They should not be used, so this is another case in which you can erase the unused elements** (line 35). You also may provide a **binary predicate function** specifying how to compare two elements for equality.

```
32     // eliminate duplicate values from v
33     std::vector v(results2.begin(), results2.end());
34     auto [first, last]{std::ranges::unique(v)};
35     v.erase(first, last); // remove elements that no longer contain values
36
37     std::cout << "\n\nAfter unique v contains: ";
38     std::ranges::copy(v, output);
39
After unique, v contains: 1 2 3 4 5 7 9
```

`reverse` Algorithm

20 Concepts The **C++20** **`std::ranges::reverse`** **algorithm** (line 41) reverses the elements in its argument—a **`bidirectional_range`** that supports **`bidirectional_iterators`**.

```
40     std::cout << "\n\nAfter reverse, a1 contains: ";
41     std::ranges::reverse(a1); // reverse elements of a1
42     std::ranges::copy(a1, output);
43
After reverse, a1 contains: 9 7 5 3 1
```

`copy_if` Algorithm

11 20 Concepts C++11 added the algorithms **`copy_if`** and **`copy_n`**, and C++20 added **ranges** versions of both. The **C++20** **`std::ranges::copy_if`** **algorithm** (lines 47–48) receives as arguments an **`input_range`**, an **output iterator** and a **unary predicate function**. The algorithm calls the **unary predicate function** for each element in the **`input_range`** and copies only those elements for which the function returns **`true`**. The **output iterator** specifies where to output elements—in this case, we use a **`back_inserter`** to add elements to a **`vector`**. The algorithm returns a **`std::ranges::in_out_result`** containing two iterators:

- the first is an **input iterator** positioned at the end of the **`input_range`**, and

- the second is an **output iterator** positioned after the last element copied into the output container.

```
44    // copy odd elements of a2 into v2
45    std::vector<int> v2{};
46    std::cout << "\n\nAfter copy_if, v2 contains: ";
47    std::ranges::copy_if(a2, std::back_inserter(v2),
48       [](auto x){return x % 2 == 0;});
49    std::ranges::copy(v2, output);
50
After copy_if, v2 contains: 2 4
```

`copy_n` Algorithm

20 Concepts The **C++20** **`std::ranges::copy_n`** **algorithm** (line 54) copies from the location specified by the **`input_iterator`** in its first argument (`a2.begin()`) the number of elements specified by its second argument (`3`). The elements are output to the location specified by the **output iterator** in the third argument—in this case, we use a **`back_inserter`** to add elements to a `vector`.

```
51    // copy three elements of a2 into v3
52    std::vector<int> v3{};
53    std::cout << "\n\nAfter copy_n, v3 contains: ";
54    std::ranges::copy_n(a2.begin(), 3, std::back_inserter(v3));
55    std::ranges::copy(v3, output);
56    std::cout << "\n";
57 }
After copy_n, v3 contains: 2 4 5
```

14.4.9 `inplace_merge`, `unique_copy` and `reverse_copy`

20 Figure 14.10 demonstrates algorithms **`inplace_merge`**, **`unique_copy`** and **`reverse_copy`** from **C++20's** **`std::ranges`** **namespace**.

```
 1 // fig14_10.cpp
 2 // Algorithms inplace_merge, unique_copy and reverse_copy.
 3 #include <algorithm>
 4 #include <array>
 5 #include <iostream>
 6 #include <iterator>
 7 #include <vector>
 8
 9 int main() {
10    std::array a1{1, 3, 5, 7, 9, 1, 3, 5, 7, 9};
11    std::ostream_iterator<int> output{std::cout, " "};
12
13    std::cout << "array a1 contains: ";
14    std::ranges::copy(a1, output);
15
```

```
array a1 contains: 1 3 5 7 9 1 3 5 7 9
```

Fig. 14.10 Algorithms `inplace_merge`, `unique_copy` and `reverse_copy`.

`inplace_merge` Algorithm

20 Concepts Line 18 calls the **C++20** **`std::ranges::inplace_merge`** **algorithm** to merge two sorted ranges of elements in its **`bidirectional_range`** argument. This example processes the range `a1` (the first argument), merging the elements from `a1.begin()` up to, but not including, `a1.begin() + 5` (the second argument) with the elements starting from `a1.begin() + 5` (the second argument) up to, but not including the end of the range. You also can pass a **binary predicate function** that compares elements in the two subranges and returns `true` if the first should be considered less than the second.

```
16    // merge first half of a1 with second half of a1 such that
17    // a1 contains sorted set of elements after merge
18    std::ranges::inplace_merge(a1, a1.begin() + 5);
19    std::cout << "\nAfter inplace_merge, a1 contains: ";
20    std::ranges::copy(a1, output);
21
After inplace_merge, a1 contains: 1 1 3 3 5 5 7 7 9 9
```

`unique_copy` Algorithm

20 Concepts Line 24 calls the **C++20** **`std::ranges::unique_copy`** **algorithm** to copy the unique elements in its first argument's sorted **`input_range`**. The output iterator supplied as the second argument specifies where to place the copied elements—in this case, the **`back_inserter`** adds new elements in the `vector results1`, growing it as necessary. You also can pass a **binary predicate function** specifying how to compare elements for equality.

```
22    // copy only unique elements of a1 into results1
23    std::vector<int> results1{};
24    std::ranges::unique_copy(a1, std::back_inserter(results1));
25    std::cout << "\nAfter unique_copy, results1 contains: ";
26    std::ranges::copy(results1, output);
27
After unique_copy results1 contains: 1 3 5 7 9
```

`reverse_copy` Algorithm

20 Line 30 calls the **C++20** **`std::ranges::reverse_copy`** **algorithm** to make a reversed copy of its first argument's **`bidirectional_range`**. The **output iterator** supplied as the second argument specifies where to place the copied elements—in this case, the **`back_inserter`** adds new elements to the `vector results2`, growing it as necessary.

```
28    // copy elements of a1 into results2 in reverse order
29    std::vector<int> results2{};
30    std::ranges::reverse_copy(a1, std::back_inserter(results2));
31    std::cout << "\nAfter reverse_copy, results2 contains: ";
32    std::ranges::copy(results2, output);
```

```
33     std::cout << "\n";
34  }
After reverse_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1
```

14.4.10 Set Operations

Figure 14.11 demonstrates the **C++20 `std::ranges` namespace's** set-manipulation **algorithms `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference`** and **`set_union`**. In addition to the capabilities we show, you can customize each algorithm's element comparisons by passing a **binary predicate function** that receives two elements and returns `true` if the first should be considered less than the second.

```
1  // fig14_11.cpp
2  // Algorithms includes, set_difference, set_intersection,
3  // set_symmetric_difference and set_union.
4  #include <array>
5  #include <algorithm>
6  #include <fmt/format.h> // C++20: This will be #include <format>
7  #include <iostream>
8  #include <iterator>
9  #include <vector>
10
11 int main() {
12    std::array a1{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
13    std::array a2{4, 5, 6, 7, 8};
14    std::array a3{4, 5, 6, 11, 15};
15    std::ostream_iterator<int> output{std::cout, " "};
16
17    std::cout << "a1 contains: ";
18    std::ranges::copy(a1, output); // display array a1
19    std::cout << "\na2 contains: ";
20    std::ranges::copy(a2, output); // display array a2
21    std::cout << "\na3 contains: ";
22    std::ranges::copy(a3, output); // display array a3
23
a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15
```

Fig. 14.11 Algorithms `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` and `set_union`.

`includes` Algorithm

20 Concepts Lines 26 and 30 each call the **C++20 `std::ranges::includes` algorithm** to compare two sorted **`input_ranges`** and determine whether every element of the second is in the first. The **ranges must be sorted** using the same **comparison function**. The algorithm returns `true` if all the second **range**'s elements are in the first range; otherwise, it returns `false`. In line 26, `a2`'s elements are all in `a1`, so `includes` returns `true`. In line 30, `a3`'s elements are not all in `a1`, so **`includes`** returns `false`.

```
24    // determine whether a2 is completely contained in a1
25    std::cout << fmt::format("\n\na1 {} a2",
```

```
26       std::ranges::includes(a1, a2) ? "includes" : "does not include");
27
28    // determine whether a3 is completely contained in a1
29    std::cout << fmt::format("\n\na1 {} a3",
30       std::ranges::includes(a1, a3) ? "includes" : "does not include");
31
```

```
a1 includes a2

a1 does not include a3
```

`set_difference` Algorithm

20 Concepts Line 34 calls the **C++20** **`std::ranges::set_difference`** **algorithm** to find the elements from the first sorted **`input_range`** that are not in the second sorted **`input_range`**. The ranges must be sorted using the same **comparison function**. The elements that differ are copied to the location specified by the third argument's **output iterator**—in this case, a **`back_inserter`** adds them to the `vector difference`.

```
32    // determine elements of a1 not in a2
33    std::vector<int> difference{};
34    std::ranges::set_difference(a1, a2, std::back_inserter(difference));
35    std::cout << "\n\nset_difference of a1 and a2 is: ";
36    std::ranges::copy(difference, output);
37
```

```
set_difference of a1 and a2 is: 1 2 3 9 10
```

`set_intersection` Algorithm

20 Concepts Lines 40–41 call the **C++20** **`std::ranges::set_intersection`** **algorithm** to determine the elements from the first sorted **`input_range`** that are in the second sorted **`input_range`**. The ranges must be sorted using the same **comparison function**. The elements common to both are copied to the location specified by the third argument's **output iterator**—in this case, a **`back_inserter`** adds them to the `vector intersection`.

```
38    // determine elements in both a1 and a2
39    std::vector<int> intersection{};
40    std::ranges::set_intersection(a1, a2,
41       std::back_inserter(intersection));
42    std::cout << "\n\nset_intersection of a1 and a2 is: ";
43    std::ranges::copy(intersection, output);
44
```

```
set_intersection of a1 and a2 is: 4 5 6 7 8
```

`set_symmetric_difference` Algorithm

20 Concepts Lines 48–49 call the **C++20** **`std::ranges::set_symmetric_difference`** **algorithm** to determine the elements in the first sorted **`input_range`** that are not in the second sorted **`input_range`** and the elements in the second that are not in the first. The ranges must be sorted using the same **comparison function**. Each **`input_range`**'s elements that are different are copied to the location

specified by the third argument's **output iterator**—in this case, a `back_inserter` adds them to `symmetricDifference`.

```
45     // determine elements of a1 that are not in a3 and
46     // elements of a3 that are not in a1
47     std::vector<int> symmetricDifference{};
48     std::ranges::set_symmetric_difference(a1, a3,
49        std::back_inserter(symmetricDifference));
50     std::cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
51     std::ranges::copy(symmetricDifference, output);
52
set_symmetric_difference of a1 and a3 is: 1 2 3 7 8 9 10 11 15
```

`set_union` Algorithm

20 Concepts Line 55 calls the **C++20** `std::ranges::set_union` **algorithm** to create a set of the elements in either or both of its two sorted `input_range`s, which must be sorted using the same **comparison function**. The elements are copied to the location specified by the third argument's **output iterator**—in this case, a `back_inserter` adds them to `unionSet`. Elements that appear in both sets are copied only from the first set.

```
53     // determine elements that are in either or both sets
54     std::vector<int> unionSet{};
55     std::ranges::set_union(a1, a3, std::back_inserter(unionSet));
56     std::cout << "\n\nset_union of a1 and a3 is: ";
57     std::ranges::copy(unionSet, output);
58     std::cout << "\n";
59  }
set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15
```

14.4.11 `lower_bound`, `upper_bound` and `equal_range`

20 Figure 14.12 demonstrates the algorithms `lower_bound`, `upper_bound` and `equal_range` from **C++20's** `std::ranges` **namespace**.[23] In addition to the capabilities we show, you can customize each algorithm's element comparisons by passing a **binary predicate function** that receives two elements and returns `true` if the first should be considered less than the second.

23. As of January 2022, `std::ranges::equal_range` does not compile using the most recent `clang++`.

```
1  // fig14_12.cpp
2  // Algorithms lower_bound, upper_bound and
3  // equal_range for a sorted sequence of values.
4  #include <algorithm>
5  #include <array>
6  #include <iostream>
7  #include <iterator>
8
9  int main() {
```

```
10     std::array values{2, 2, 4, 4, 4, 6, 6, 6, 6, 8};
11     std::ostream_iterator<int> output{std::cout, " "};
12
13     std::cout << "values contains: ";
14     std::ranges::copy(values, output);
15
values contains: 2 2 4 4 4 6 6 6 6 8
```

Fig. 14.12 Algorithms `lower_bound`, `upper_bound` and `equal_range` for a sorted sequence of values.

`lower_bound` Algorithm

20 Concepts Line 17 calls the **C++20** **`std::ranges::lower_bound`** **algorithm** to find in a **sorted** **`forward_range`** the first location at which the second argument could be inserted such that the range would still be **sorted in ascending order**. The algorithm returns an iterator pointing to that location.

```
16     // determine lower-bound insertion point for 6 in values
17     auto lower{std::ranges::lower_bound(values, 6)};
18     std::cout << "\n\nLower bound of 6 is index: "
19        << (lower - values.begin());
20
Lower bound of 6 is index: 5
```

`upper_bound` Algorithm

20 Concepts Line 22 calls the **C++20** **`std::ranges::upper_bound`** **algorithm** to find in a **sorted** **`forward_range`** the last location at which the second argument could be inserted such that the range would still be **sorted in ascending order**. The algorithm returns an iterator pointing to that location.

```
21     // determine upper-bound insertion point for 6 in values
22     auto upper{std::ranges::upper_bound(values, 6)};
23     std::cout << "\nUpper bound of 6 is index: "
24        << (upper - values.begin());
25
Upper bound of 6 is index: 9
```

`equal_range` Algorithm

20 The **C++20** **`std::ranges::equal_range`** **algorithm** (line 27) performs the **`lower_bound`** and **`upper_bound`** operations, then returns their results as a `std::ranges::subrange`, which we unpack into variables `first` and `last`.

```
26     // use equal_range to determine the lower and upper bound of 6
27     auto [first, last]{std::ranges::equal_range(values, 6)};
28     std::cout << "\nUsing equal_range:\n Lower bound of 6 is index: "
29        << (first - values.begin());
30     std::cout << "\n Upper bound of 6 is index: "
31        << (last - values.begin());
```

```
32
Using equal_range:
   Lower bound of 6 is index: 5
   Upper bound of 6 is index: 9
```

Locating Insertion Points in Sorted Sequences

Algorithms **`lower_bound`**, **`upper_bound`** and **`equal_range`** are often used to locate a new value's insertion point in a sorted sequence. Line 36 uses **`lower_bound`** to locate the first position where `3` can be inserted in order in `values`. Line 43 uses **`upper_bound`** to locate the last point where `7` can be inserted in order in `values`.

```
33    // determine lower-bound insertion point for 3 in values
34    std::cout << "\n\nUse lower_bound to locate the first point "
35       << "at which 3 can be inserted in order";
36    lower = std::ranges::lower_bound(values, 3);
37    std::cout << "\n Lower bound of 3 is index: "
38       << (lower - values.begin());
39
40    // determine upper-bound insertion point for 7 in values
41    std::cout << "\n\nUse upper_bound to locate the last point "
42       << "at which 7 can be inserted in order";
43    upper = std::ranges::upper_bound(values, 7);
44    std::cout << "\n Upper bound of 7 is index: "
45       << (upper - values.begin()) << "\n";
46 }
Use lower_bound to locate the first point at which 3 can be inserted in order
   Lower bound of 3 is index: 2

Use upper_bound to locate the last point at which 7 can be inserted in order
   Upper bound of 7 is index: 9
```

14.4.12 `min`, `max` and `minmax`

20 Figure 14.13 demonstrates algorithms **`min`**, **`max`** and **`minmax`** from the `std` namespace and the **`minmax`** overload from **C++20's `std::ranges` namespace**. Unlike the algorithms we presented in Section 14.4.5, which operated on **ranges**, the `std` namespace's **`min`**, **`max`** and **`minmax`** algorithms operate on two values passed as arguments. The **`std::ranges::minmax`** algorithm returns the minimum and maximum values in a **range**.

Algorithms `min` and `max` with Two Parameters

The **`min`** and **`max` algorithms** (lines 8–12) each receive two arguments and return the minimum or maximum value. Each algorithm also has an overload that takes as a third argument a **binary predicate function** for **custom comparisons** determining whether the first argument should be considered less than the second.

```
1  // fig14_13.cpp
2  // Algorithms min, max and minmax.
3  #include <array>
4  #include <algorithm>
```

```
 5  #include <iostream>
 6
 7  int main() {
 8     std::cout << "Minimum of 12 and 7 is: " << std::min(12, 7)
 9        << "\nMaximum of 12 and 7 is: " << std::max(12, 7)
10        << "\nMinimum of 'G' and 'Z' is: '" << std::min('G', 'Z') << "'"
11        << "\nMaximum of 'G' and 'Z' is: '" << std::max('G', 'Z') << "'"
12        << "\nMinimum of 'z' and 'Z' is: '" << std::min('z', 'Z') << "'";
13
```

```
Minimum of 12 and 7 is: 7
Maximum of 12 and 7 is: 12
Minimum of 'G' and 'Z' is: 'G'
Maximum of 'G' and 'Z' is: 'Z'
Minimum of 'z' and 'Z' is: 'Z'
```

Fig. 14.13 Algorithms `min`, `max` and `minmax`.

C++11 `minmax` Algorithm with Two Arguments

11 C++11 added the two-argument **`minmax` algorithm** (line 15), which returns a **pair of values** containing the smaller and larger items, respectively. Here we used **structured bindings** to unpack the values into `smaller` and `larger`. A second version of **`minmax`** takes as a third argument a **binary predicate function** for a **custom comparison** determining whether the first argument should be considered less than the second.

```
14     // determine which argument is the min and which is the max
15     auto [smaller, larger]{std::minmax(12, 7)};
16     std::cout << "\n\nMinimum of 12 and 7 is: " << smaller
17        << "\nMaximum of 12 and 7 is: " << larger;
18
```

```
Minimum of 12 and 7 is: 7
Maximum of 12 and 7 is: 12
```

`minmax` Algorithm for C++20 Ranges

20 Concepts The **C++20 `std::ranges::minmax` algorithm** (line 25) returns a pair of values containing the minimum and maximum items in its **`input_range`** or **`initializer_list`** argument. Again, we used **structured bindings** to unpack these values into `smallest` and `largest`, respectively. You also can pass as a second argument a **binary predicate function** for comparing elements to determine whether the first should be considered less than the second.

```
19     std::array items{3, 100, 52, 77, 22, 31, 1, 98, 13, 40};
20     std::ostream_iterator<int> output{std::cout, " "};
21
22     std::cout << "\n\nitems: ";
23     std::ranges::copy(items, output);
24
25     auto [smallest, largest]{std::ranges::minmax(items)};
26     std::cout << "\nMinimum value in items: " << smallest
27        << "\nMaximum value in items is: " << largest << "\n";
28  }
```

```
items: 3 100 52 77 22 31 1 98 13 40
```

```
Minimum value in items: 1
Maximum value in items is: 100
```

14.4.13 Algorithms `gcd`, `lcm`, `iota`, `reduce` and `partial_sum` from Header `<numeric>`

23 Section 8.19.2 introduced the **`accumulate`** algorithm (**header `<numeric>`**). Figure 14.14 demonstrates **`<numeric>` algorithms `gcd`, `lcm`, `iota`, `reduce`** and **`partial_sum`**. This header's algorithms require **common ranges** and are expected to have ranges overloads in C++23.[24]

24. Barry Revzin, Conor Hoekstra and Tim Song, "A Plan for C++23 Ranges," October 14, 2020. Accessed January 29, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2214r0.html#algorithms.

`gcd` Algorithm

The **`gcd` algorithm** (lines 14 and 15) receives two integer arguments and returns their **greatest common divisor**.

```
 1  // fig14_14.cpp
 2  // Demonstrating algorithms gcd, lcm, iota, reduce and partial_sum.
 3  #include <array>
 4  #include <algorithm>
 5  #include <functional>
 6  #include <iostream>
 7  #include <iterator>
 8  #include <numeric>
 9
10  int main() {
11     std::ostream_iterator<int> output{std::cout, " "};
12
13     // calculate the greatest common divisor of two integers
14     std::cout << "std::gcd(75, 20): " << std::gcd(75, 20)
15        << "\nstd::gcd(17, 13): " << std::gcd(75, 13);
16
```

```
std::gcd(75, 20): 5
std::gcd(17, 13): 1
```

Fig. 14.14 Demonstrating algorithms `gcd`, `lcm`, `iota`, `reduce` and `partial_sum`.

`lcm` Algorithm

The **`lcm` algorithm** (lines 18 and 19) receives two integer arguments and returns their **least common multiple**.

```
17     // calculate the least common multiple of two integers
18     std::cout << "\n\nstd::lcm(3, 5): " << std::lcm(3, 5)
19        << "\nstd::lcm(12, 9): " << std::lcm(12, 9);
20
```

```
std::lcm(3, 5): 15
std::lcm(12, 9): 36
```

`iota` Algorithm

The **`iota` algorithm** (line 23) fills a **common range** with a sequence of values starting with the value in the third argument. The first two arguments must be **forward iterators** representing a **common range** to fill. The last argument's type must support the `++` operator.

```
21    // fill an array with integers using the std::iota algorithm
22    std::array<int, 5> ints{};
23    std::iota(ints.begin(), ints.end(), 1);
24    std::cout << "\n\nints: ";
25    std::ranges::copy(ints, output);
26
```

```
ints: 1 2 3 4 5
```

`reduce` Algorithm

The **`reduce` algorithm** (lines 29 and 31) reduces a **common range**'s elements to a single value. The first and second arguments must be **input iterators**. The call in line 29 implicitly adds the **common range**'s elements. The call in line 31 provides a custom initializer value (`1`) and a **binary function** that specifies how to perform the **reduction**. In this case, we used **`std::multiplies{}`** (header **`<functional>`**)—a predefined **binary function object**[25] that multiplies its two arguments and returns the result.
The `{}` create a temporary **`std::multiplies`** object and call its constructor. Every function object has an overloaded **`operator()`** function. Inside the **`reduce`** algorithm, it calls function **`operator()`** on the function object to produce a result. Any commutative and associative binary function that takes two values of the same type and returns a result of that type can be passed as the fourth argument. In Section 14.5, you'll see that header **`<functional>`** defines **binary function objects** for addition, subtraction, multiplication, division and modulus, among other operations.

25. We say more about the predefined function objects in Section 14.5.

```
27    // reduce elements of a container to a single value
28    std::cout << "\n\nsum of ints: "
29       << std::reduce(ints.begin(), ints.end())
30       << "\nproduct of ints: "
31       << std::reduce(ints.begin(), ints.end(), 1, std::multiplies{});
32
```

```
sum of ints: 15
product of ints: 120
```

`reduce` vs. **`accumulate`**

The **`reduce`** algorithm is similar to the **`accumulate`** algorithm **but does not guarantee the order in which the elements are processed**. In Chapter 17, you'll see that **this difference in operation is why the `reduce` algorithm can be parallelized for better performance, but the `accumulate` algorithm cannot.**[26]

26. Sy Brand, "`std::accumulate` vs. `std::reduce`," May 15, 2018. Accessed January 29, 2022. https://blog.tartanllama.xyz/accumulate-vs-reduce/.

`partial_sum` Algorithm

The **`partial_sum` algorithm** (lines 37 and 39) calculates a partial sum of its **common range's** elements from the start of the range through the current element. By default, this version of **`partial_sum`** uses the **`std::plus` function object**, which adds its two arguments and returns their sum. For `ints`, which line 23 filled with the values 1, 2, 3, 4 and 5, the call in line 37 outputs the following sums:

- 1 (this is simply the value of `ints`' first element)
- 3 (the sum 1 + 2)
- 6 (the sum 1 + 2 + 3)
- 10 (the sum 1 + 2 + 3 + 4)
- 15 (the sum 1 + 2 + 3 + 4 + 5)

```
33    // calculate the partial sums of ints' elements
34    std::cout << "\n\nints: ";
35    std::ranges::copy(ints, output);
36    std::cout << "\n\npartial_sum of ints using std::plus by default: ";
37    std::partial_sum(ints.begin(), ints.end(), output);
38    std::cout << "\npartial_sum of ints using std::multiplies: ";
39    std::partial_sum(ints.begin(), ints.end(), output, std::multiplies{});
40    std::cout << "\n";
41 }
```

```
ints: 1 2 3 4 5

partial_sum of ints using std::plus by default: 1 3 6 10 15
partial_sum of ints using std::multiplies: 1 2 6 24 120
```

The second call (line 39) uses the **`partial_sum`** overload that receives a **binary function** specifying how to perform partial calculations. In this case, we used the predefined **binary function object `std::multiplies{}`**, resulting in the products of the values from the beginning of the container to the current element:

- 1 (this is simply the value of `ints`' first element)
- 2 (the product 1 * 2)
- 6 (the product 1 * 2 * 3)
- 24 (the product 1 * 2 * 3 * 4)
- 120 (the product 1 * 2 * 3 * 4 * 5)

This `partial_sum` call is actually calculating the factorials of 1–5 (that is 1!, 2!, 3!, 4! and 5!).

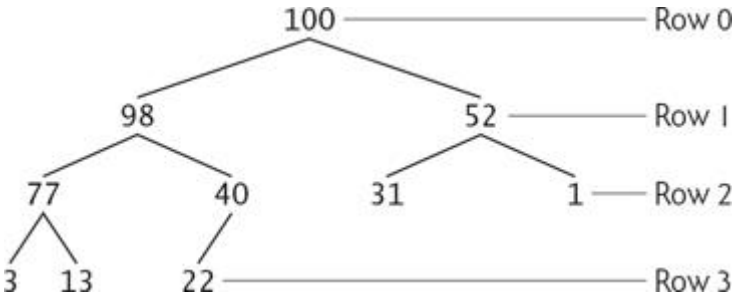
14.4.14 Heapsort and Priority Queues

Section 13.10.3 introduced the **`priority_queue` container adaptor**. Elements added to a **`priority_queue`** are stored in a manner that enables removing them in **priority order**. The highest-priority element is always removed first—usually, the highest-priority element has the largest value, but this is customizable. Finding the highest-priority element can be accomplished efficiently by arranging the elements in a data structure called a **heap**—not to be confused with the heap C++ maintains for dynamic memory allocation. A common use of priority queues is in operating system process scheduling.

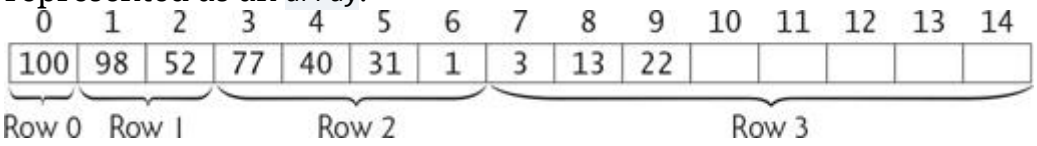
Heap Data Structure

A **heap** is commonly implemented as a **binary tree**. A **max heap** stores its largest value in the root node, and **any given child node's value is less than or equal to its parent node's value**. Heaps may

contain duplicate values. A **heap** also can be a **min heap** in which the smallest value is in the root node, and any given child node's value is greater than or equal to its parent node's value. The following diagram shows a binary tree representing a **max heap**:



A **heap** is typically stored in an array-like data structure, such as an `array`, `vector` or `deque`, each of which uses **random-access iterators**. The following diagram shows the preceding diagram's max heap represented as an `array`:



You can confirm that this array represents a max heap. For any given array index `n`, you can find the parent node's array index by calculating

```
(n - 1) / 2
```

using integer arithmetic. Then you can confirm that the child node's value is less than or equal to the parent node's value. A **max heap**'s largest value is always at the top of the binary tree, which corresponds to the `array` element at index `0`.

Heap-Related Algorithms

20 Figure 14.15 demonstrates four **C++20** `std::ranges` **namespace** algorithms related to **heaps**. First, we show `make_heap` and `sort_heap`, which implement the two steps in the **heapsort algorithm**, which has a worst-case runtime of $O(n \log n)$:[27]

27. "Heapsort." Wikipedia. Wikimedia Foundation. Accessed January 29, 2022. https://en.wikipedia.org/wiki/Heapsort.

- *Step 1* arranges the elements of a container into a **heap**.
- *Step 2* removes the elements from the **heap** to produce a sorted sequence.

```
1  // fig14_15.cpp
2  // Algorithms make_heap, sort_heap, push_heap and pop_heap.
3  #include <iostream>
4  #include <algorithm>
5  #include <array>
6  #include <vector>
7  #include <iterator>
8
9  int main() {
10    std::ostream_iterator<int> output{std::cout, " "};
11
```

Fig. 14.15 Algorithms `make_heap`, `sort_heap`, `push_heap` and `pop_heap`.

Then, we show `push_heap` and `pop_heap`, which the `priority_queue` **container adaptor**[28] uses "under the hood" to maintain its elements in a **heap** as elements are added and removed.

28. You can see the open-source Microsoft C++ standard library implementation of `priority_queue` using `push_heap` and `pop_heap` at https://github.com/microsoft/STL/blob/main/stl/inc/queue. `Accessed January 29, 2022`.

Initializing and Displaying `heapArray`

Line 12 creates and initializes the `array heapArray` with 10 different unsorted integers. Line 14 displays `heapArray` before we convert its contents to a **heap** and **sort** the elements.

```
12    std::array heapArray{3, 100, 52, 77, 22, 31, 1, 98, 13, 40};
13    std::cout << "heapArray before make_heap:\n";
14    std::ranges::copy(heapArray, output);
15
```

```
heapArray before make_heap:
3 100 52 77 22 31 1 98 13 40
```

`make_heap` Algorithm

20 Line 16 calls the **C++20** `std::ranges::make_heap` **algorithm** to arrange the elements of its `random_access_range` argument into a **heap**, which can then be used with `sort_heap` to produce a sorted sequence. Line 18 displays `heapArray` with its elements arranged in a **heap**.

A `random_access_range` supports `random_access_iterator`s, so this algorithm works with `array`s, `vector`s and `deque`s.

```
16    std::ranges::make_heap(heapArray); // create heap from heapArray
17    std::cout << "\nheapArray after make_heap:\n";
18    std::ranges::copy(heapArray, output);
19
```

```
heapArray after make_heap:
100 98 52 77 40 31 1 3 13 22
```

`sort_heap` Algorithm

20 Line 20 calls the **C++20** `std::ranges::sort_heap` **algorithm** to sort the elements of its `random_access_range` argument. The range must already be a **heap**. Line 22 displays the sorted `heapArray`.

```
20    std::ranges::sort_heap(heapArray); // sort elements with sort_heap
21    std::cout << "\nheapArray after sort_heap:\n";
22    std::ranges::copy(heapArray, output);
23
```

```
heapArray after sort_heap:
1 3 13 22 31 40 52 77 98 100
```

Using `push_heap` and `pop_heap` to Maintain a Heap

Next, we'll demonstrate the algorithms a `priority_queue` uses "under the hood" to **insert a new item in a heap** and **remove an item from a heap**. Both operations are $O(\log n)$.[29] Lines 25–33 define the lambda `push`, which adds one `int` value to a heap that's stored in a `vector`. To do so, `push` performs the following tasks:

29. "Binary Heap." Wikipedia. Wikimedia Foundation. Accessed January 29, 2022. https://en.wikipedia.org/wiki/Binary_heap.

- Line 28 appends one `int` to the `vector` argument `heap`.
- 20 Line 29 calls the **C++20** `std::ranges::push_heap` **algorithm**, which takes the last element of its `random_access_range` argument (`heap`) and inserts it into the **heap data structure**. Each time `push_heap` is called, it assumes that the last element is being added to the **heap** and that the other elements are already arranged as a heap. If the element appended in line 28 is the range's only element, the range is already a heap. Otherwise, `push_heap` rearranges the elements into a heap.
- Line 31 displays the updated **heap data structure** after each value is added.

```
24    // lambda to add an int to a heap
25    auto push{
26       [&](std::vector<int>& heap, int value) {
27          std::cout << "\n\npushing " << value << " onto heap";
28          heap.push_back(value); // add value to the heap
29          std::ranges::push_heap(heap); // insert last element into heap
30          std::cout << "\nheap: ";
31          std::ranges::copy(heap, output);
32       }
33    };
34
```

Lines 36–44 define the lambda `pop`, which removes the largest value from the heap data structure. To do so, `pop` performs the following tasks:

- 20 Line 38 calls the **C++20** `std::ranges::pop_heap` **algorithm** to remove the heap's largest value. The algorithm assumes that its `random_access_range` represents a **heap data structure**. First, it swaps the largest heap element (located at `heap.begin()`) with the last heap element (the one before `heap.end()`). Then, it ensures that the elements from the range's beginning up to, but not including, the range's last element still form a heap. **The `pop_heap` algorithm does not modify the number of elements in the range.**
- Line 39 displays the value of the `vector`'s last element—the value that was just removed from the heap but still remains in the `vector`.
- Line 40 removes the `vector`'s last element, leaving only the `vector` elements that still represent a **heap data structure**.
- Line 42 shows the current **heap data structure** contents.

```
35    // lambda to remove an item from the heap
36    auto pop{
37       [&](std::vector<int>& heap) {
38          std::ranges::pop_heap(heap); // remove max item from heap
39          std::cout << "\n\npopping highest priority item: " << heap.back();
40          heap.pop_back(); // remove vector's last element
41          std::cout << "\nheap: ";
42          std::ranges::copy(heap, output);
```

```
43        }
44     };
45
```

Demonstrating a Heap Data Structure

Line 46 defines an empty `vector<int>` in which we'll maintain the **heap data structure**. Lines 49–51 call the **`push` lambda** to add the values `3`, `52` and `100` to the heap. As we add each value, note that the largest value is always stored in the **`vector`**'s first element and that the elements are not stored in sorted order.

```
46     std::vector<int> heapVector{};
47
48     // place five integers into heapVector, maintaining it as a heap
49     for (auto value : {3, 52, 100}) {
50        push(heapVector, value);
51     }
52
```

```
pushing 3 onto heap
heap: 3

pushing 52 onto heap
heap: 52 3

pushing 100 onto heap
heap: 100 3 52
```

Next, line 53 calls the **`pop` lambda** to remove the highest-priority item (`100`) from the heap. Note that the largest remaining value (`52`) is now in the **`vector`**'s first element. Line 54 adds the value `22` to the heap.

```
53     pop(heapVector); // remove max item
54     push(heapVector, 22); // add new item to heap
55
```

```
popping highest priority item: 100
heap: 52 3

pushing 22 onto heap
heap: 52 3 22
```

Next, line 56 removes the highest-priority item (`52`) from the heap. Again, the largest remaining value (`22`) is now in the **`vector`**'s first element. Line 57 adds the value `77` to the heap. This is now the largest value, so it becomes the **`vector`**'s first element.

```
56     pop(heapVector); // remove max item
57     push(heapVector, 77); // add new item to heap
58
```

```
popping highest priority item: 52
heap: 22 3

pushing 77 onto heap
```

```
heap: 77 3 22
```

Finally, lines 59–61 remove the heap's three remaining items. Note that after line 59 executes, the largest remaining element (22) becomes the `vector`'s first element.

```
59     pop(heapVector); // remove max item
60     pop(heapVector); // remove max item
61     pop(heapVector); // remove max item
62     std::cout << "\n";
63 }
```

```
popping highest priority item: 77
heap: 22 3

popping highest priority item: 22
heap: 3

popping highest priority item: 3
heap:
```

14.5 Function Objects (Functors)

SE As we've shown, many standard library algorithms allow you to pass a lambda or a function pointer into the algorithm to help it perform its task. Any algorithm that can receive a lambda or function pointer can also receive a **function object** (also called a **functor**)—that is, an object of a class that overloads the function-call operator (parentheses) with a function named `operator()`. The overloaded `operator()` must meet the algorithm's requirements for the number of parameters and the return type.

Function objects can be used syntactically and semantically like a lambda or a function pointer. The `operator()` function is invoked using the **object's name followed by parentheses containing the arguments**. Most algorithms can use lambdas, function pointers and function objects interchangeably.

Benefits of Function Objects

Function objects provide several benefits over functions and pointers to functions:

- A **function object** is an object of a class, so it can contain non-`static` data to maintain state for a specific **function object** or `static` data to maintain state shared by all function objects of that class type. Also, the class type of a **function object** can be used as a default type argument for template type parameters.[30]

30. "Function Objects in the C++ Standard Library," March 15, 2019. Accessed January 29, 2022. https://docs.microsoft.com/en-us/cpp/standard-library/function-objects-in-the-stl.

- Perf Perhaps most importantly, **the compiler can inline function objects** for performance. A **function object's** `operator()` function typically is defined in its class's body, making it implicitly `inline`. When defined outside its class's body, function `operator()` can be declared `inline` explicitly. **Compilers implement lambdas as function objects, so these, too, can be inlined**. On the other hand, compilers typically do not inline functions invoked via function pointers—such pointers could be aimed at any function with the appropriate parameters and return type, so a compiler does not know which function to inline.[31]

31. Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. p.201–202: Pearson Education, 2001.

Predefined Function Objects of the Standard Library

20 Header `<functional>` contains many **predefined function objects**. Each is implemented as a class template. The following table lists some of the commonly used standard library **function objects**. Each **relational and equality function object** has a corresponding one of the same name in the **C++20** `std::ranges` **namespace**. Most are **binary function objects** that receive two arguments and return a result. Both `logical_not` and `negate` are **unary function objects** that receive one argument and return a result.

Function object	Type
`divides<T>`	arithmetic
`equal_to<T>`	relational
`greater<T>`	relational
`greater_equal<T>`	relational
`less<T>`	relational
`less_equal<T>`	relational
`logical_and<T>`	logical
`logical_not<T>`	logical
`logical_or<T>`	logical
`minus<T>`	arithmetic
`modulus<T>`	arithmetic
`negate<T>`	arithmetic
`not_equal_to<T>`	relational
`plus<T>`	arithmetic
`multiplies<T>`	Arithmetic

Consider the following function object for comparing two `int` values:

```
std::less<int> smaller{};
```

We can use the object's name (`smaller`) to call its **`operator()` function**, as follows:

```
smaller(10, 7)
```

Here, `smaller` would return `false` because `10` is not less than `7`. An algorithm like `sort` would use this information to reorder these values into ascending order. We used the function object **`less<T>`** in Section 13.9's presentations to specify the order for keys in the ordered set and map containers.

You can see the complete list of **function objects** at

```
https://en.cppreference.com/w/cpp/utility/functional
```

and in the "Function Objects" section of the C++ standard.[32] The **`std::ranges` algorithms** that compare elements for ordering use **`less<T>`** as their **default predicate function argument**. Recall that many of the overloaded standard library algorithms that perform comparisons can receive a **binary function** that determines whether its first argument is less than its second—precisely the purpose of the **`less<T>` function object**.

32. "General Utilities Library—function Objects." Accessed January 29, 2022. https://timsongcpp.github.io/cppwp/n4861/function.objects.

Using the `accumulate` Algorithm

Figure 14.16 uses the **`std::accumulate`** numeric algorithm (**header `<numeric>`**) to calculate the sum of the squares of an `array`'s elements. The **`<numeric>`** algorithms do not have **C++20 `std::ranges` overloads**, so they use **common ranges**—**`std::ranges` overloads** of these algorithms are proposed for C++23.[33] The **`accumulate`** algorithm has two overloads. The three-argument version adds the **common range**'s elements by default. The four-argument version receives as its last argument a **binary function** that customizes how to perform the calculation. That argument can be supplied as:

33. Christopher Di Bella, "A Concept Design for the Numeric Algorithms," August 2, 2019. Accessed January 29, 2022. http://wg21.link/p1813r0.

- a **function pointer** to a **binary function** that takes two parameters of the **common range**'s element type and returns a result of that type,
- a **binary function object** in which the **`operator()`** function takes two parameters of the **common range**'s element type and returns a result of that type, or
- a **lambda** that takes two parameters of the **common range**'s element type and returns a result of that type.

This example calls **`accumulate`** with a **function pointer**, then a **function object** and then a **lambda**.

```
1  // fig14_16.cpp
2  // Demonstrating function objects.
3  #include <array>
4  #include <algorithm>
5  #include <functional>
6  #include <iostream>
```

```
 7  #include <iterator>
 8  #include <numeric>
 9
```

Fig. 14.16 Demonstrating function objects.

Function `sumSquares`

Lines 12–14 define a **function** **`sumSquares`** that takes two arguments of the same type and returns a value of that type—the requirements for the **binary function** that **`accumulate`** can receive as an argument. The `sumSquares` function returns the sum of its first argument `total` and the square of its second argument `value`.

```
10  // binary function returns the sum of its first argument total
11  // and the square of its second argument value
12  int sumSquares(int total, int value) {
13     return total + value * value;
14  }
15
```

Class `SumSquaresClass`

Lines 19–25 define class `SumSquaresClass`.[34] Its overloaded **`operator()`** has two `int` parameters and returns an `int`. This meets the requirements for the **binary function** that `accumulate` can call when processing a **common range** of `int`s. Function **`operator()`** returns the sum of its first argument `total` and the square of its second argument `value`.

34. This class handles only `int` values, but could be implemented as a class template that handles many types—we'll define custom class templates in Chapter 15.

```
16  // class SumSquaresClass defines overloaded operator()
17  // that returns the sum of its first argument total
18  // and the square of its second argument value
19  class SumSquaresClass {
20  public:
21     // add square of value to total and return result
22     int operator()(int total, int value) {
23        return total + value * value;
24     }
25  };
26
```

Calling Algorithm `accumulate`

We call **`accumulate`** three times:

- Lines 36–37 call **`accumulate`** with a **pointer to function** `sumSquares` as its last argument.
- Lines 44–45 call **`accumulate`** with a **`SumSquaresClass`** **function object** as the last argument. **`SumSquaresClass{}`** in line 45 creates a **temporary** **`SumSquaresClass`** **object** and calls its constructor. That **function object** is then passed to **`accumulate`**, which calls the **`SumSquaresClass`** **object's** **`operator()`** **function**.
- Lines 50–51 call **`accumulate`** with an equivalent **lambda**. The **lambda** performs the same tasks as the **function** **`sumSquares`** and the overloaded **`operator()`** **function in** **`SumSquaresClass`**.

```
27  int main() {
28     std::array integers{1, 2, 3, 4};
29     std::ostream_iterator<int> output{std::cout, " "};
30
31     std::cout << "array integers contains: ";
32     std::ranges::copy(integers, output);
33
34     // calculate sum of squares of elements of array integers
35     // using binary function sumSquares
36     int result{std::accumulate(integers.cbegin(), integers.cend(),
37        0, sumSquares)};
38
39     std::cout << "\n\nSum of squares\n"
40        << "via binary function sumSquares: " << result;
41
42     // calculate sum of squares of elements of array integers
43     // using binary function object
44     result = std::accumulate(integers.cbegin(), integers.cend(),
45        0, SumSquaresClass{});
46
47     std::cout << "\nvia a SumSquaresClass function object: " << result;
48
49     // calculate sum of squares array
50     result = std::accumulate(integers.cbegin(), integers.cend(),
51        0, [](auto total, auto value){return total + value * value;});
52
53     std::cout << "\nvia a lambda: " << result << "\n";
54  }
```

```
array integers contains: 1 2 3 4

Sum of squares
via binary function sumSquares: 30
via a SumSquaresClass function object: 30
via a lambda: 30
```

Each call to **`accumulate`** uses its function argument as follows:

- On the first call to its function argument, **`accumulate`** passes its third argument's value (`0` in this example) and the value of `integers`' first element (`1` in this example). This calculates and returns the result of `0 + 1 * 1`, which is `1`.
- On the second call to its function argument, **`accumulate`** passes the prior result (`1`) and the value of `integers`' next element (`2`). This calculates and returns the result of `1 + 2 * 2`, which is `5`.

- On the third call to its function argument, `accumulate` passes the prior result (`5`) and the value of `integers`' next element (`3`). This calculates and returns the result of `5 + 3 * 3`, which is `14`.
- On the last call to its function argument, `accumulate` passes the prior result (`14`) and the value of `integers`' next element (`4`). This calculates and returns the result of `14 + 16`, which is `30`.

At this point, `accumulate` reaches the end of the **common range** specified by its first two arguments, so it returns the result (`30`) of the last call to its function argument.

14.6 Projections

20 When working on objects containing multiple data items, each **C++20 `std::ranges` algorithm** can use a **projection** to select a narrower part of each object to process. Consider `Employee` objects that each have a first name, a last name and a salary. Rather than sorting `Employee`s based on all three data members, **you can sort them using only their salaries**. Figure 14.17 sorts an `array` of `Employee` objects by salary—first in **ascending order**, then in **descending order**. Lines 11–22 define class `Employee`. Lines 25–29 provide an **overloaded `operator<<` function** for `Employee`s to output them conveniently.

```
1  // fig14_17.cpp
2  // Demonstrating projections with C++20 range algorithms.
3  #include <array>
4  #include <algorithm>
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <iterator>
8  #include <string>
9  #include <string_view>
10
11 class Employee {
12 public:
13    Employee(std::string_view first, std::string_view last, int salary)
14       : m_first{first}, m_last{last}, m_salary{salary} {}
15    std::string getFirst() const {return m_first;}
16    std::string getLast() const {return m_last;}
17    int getSalary() const {return m_salary;}
18 private:
19    std::string m_first;
20    std::string m_last;
21    int m_salary;
22 };
23
24 // operator<< for an Employee
25 std::ostream& operator<<(std::ostream& out, const Employee& e) {
26    out << fmt::format("{:10}{:10}{}",
27       e.getLast(), e.getFirst(), e.getSalary());
28    return out;
29 }
30
```

Fig. 14.17 Demonstrating projections with C++20 range algorithms.

Defining and Displaying an `array<Employee>`

Lines 32–36 define an `array<Employee>`, initializing it with three `Employee`s. Line 41 displays the `Employee`s, so we can confirm later that they're sorted properly.

```
31 int main() {
32    std::array employees{
33       Employee{"Jason", "Red", 5000},
34       Employee{"Ashley", "Green", 7600},
35       Employee{"Matthew", "Indigo", 3587}
36    };
37
38    std::ostream_iterator<Employee> output{std::cout, "\n"};
39
40    std::cout << "Employees:\n";
41    std::ranges::copy(employees, output);
42
```

```
Employees:
Red       Jason       5000
Green     Ashley      7600
Indigo    Matthew     3587
```

Using a Projection to Sort the `array<Employee>` in Ascending Order

20 Lines 45–46 call the **`std::ranges::sort` algorithm**, passing three arguments:

- The first argument (`employees`) is the range to sort.
- The second argument (`{}`) is the **binary predicate function** that `sort` uses to compare elements when determining their **sort order**. The notation `{}` indicates that `sort` should use the **default binary predicate function** specified in `sort`'s definition—that is, **`std::ranges::less`**. This causes `sort` to arrange the elements in **ascending order**. The **`less` function object** compares its two arguments and returns `true` if the first is less than the second. If `less` returns `false`, the two salaries are not in ascending order, so `sort` reorders the corresponding `Employee` objects.
- The last argument specifies the **projection**. This **unary function** receives an element from the range and returns a portion of that element. Here, we implemented the unary function as a lambda that returns its `Employee` argument's salary. The **projection** is applied *before* `sort` compares the elements, so rather than comparing entire `Employee` objects to determine their sort order, `sort` compares only the `Employee`s' salaries.

```
43    // sort Employees by salary; {} indicates that the algorithm should
44    // use its default comparison function
45    std::ranges::sort(employees, {},
46       [](const auto& e) {return e.getSalary();});
47    std::cout << "\nEmployees sorted in ascending order by salary:\n";
48    std::ranges::copy(employees, output);
49
```

```
Employees sorted in ascending order by salary:
Indigo    Matthew   3587
Red       Jason     5000
Green     Ashley    7600
```

Shorthand Notation for a Projection

We can replace the unary function that we implemented as a lambda in line 46 with the shorthand notation

```
&Employee::getSalary
```

This creates a pointer to the `Employee` class's `getSalary` member function, shortening lines 45–46 to

```
std::ranges::sort(employees, {}, &Employee::getSalary);
```

To be used as a projection, the member function must be `public` and must not be overloaded. Also, it must have no parameters because the **`std::ranges` algorithms** cannot receive additional arguments to pass to the member function specified in the **projection**.[35]

35. "Under the hood," `std::ranges::sort` uses the **`std::invoke`** function (header `<functional>`) to call the provided comparator on each `Employee` object.

A Projection Can Be a Pointer to a `public` Data Member

If a class has a `public` data member, you can pass a pointer to it as the **projection argument**. For example, if our `Employee` class had a `public` data member named `salary`, we could specify `sort`'s **projection argument** as

```
&Employee::salary
```

Using a Projection to Sort the `array<Employee>` in Descending Order

In algorithms like `sort` that have function arguments, you can combine custom functions and **projections**. For example, lines 51–52 specify both a **binary predicate function object** and a **projection** to **sort** `Employee`s in **descending order** by `salary`. Again, we pass three arguments:

- The first argument (`employees`) is the range to sort.
- The second argument creates a **`std::ranges::greater` function object**. This causes **`sort`** to arrange the elements in **descending order**. The **`greater` function object** compares its two arguments and returns `true` if the first is greater than the second. If **`greater`** returns `false`, the two salaries are not in descending order, so **`sort`** reorders the corresponding `Employee` objects.
- The last argument is the **projection**. In this call, **`std::ranges::greater`** will compare the `int` salaries of `Employee`s to determine the sort order.

```
50    // sort Employees by salary in descending order
51    std::ranges::sort(employees, std::ranges::greater{},
52       &Employee::getSalary);
53    std::cout << "\nEmployees sorted in descending order by salary:\n";
54    std::ranges::copy(employees, output);
55 }
```

```
Employees sorted in descending order by salary:
Green     Ashley    7600
Red       Jason     5000
Indigo    Matthew   3587
```

20 14.7 C++20 Views and Functional-Style Programming

In Section 6.14.3, we showed **views** performing operations on ranges. We showed that **views** are **composable**, so you can **chain them together** to process a range's elements through a **pipeline of operations**. A **view** does not have its own copy of a range's elements—it simply moves

the elements through a **pipeline of operations**. **Views** are one of **C++20's key functional-style programming** capabilities.

Perf The algorithms we've presented in this chapter so far are **greedy**—when you call an algorithm, it immediately performs its specified task. You saw in Section 6.14.3 that views are **lazy**—they do not produce results until you iterate over them in a loop or pass them to an algorithm that iterates over them. As we discussed in Section 6.14.3, **lazy evaluation produces values on demand**, which can reduce your program's memory consumption and improve performance when all the values are not needed at once.

14.7.1 Range Adaptors

Section 6.14.3 also demonstrated functional-style **filter** and **map** operations using

- `std::views::filter` to keep only those **view** elements for which a **predicate function** returns `true`, and
- `std::views::transform` to **map** each **view** element to a new value, possibly of a different type.

These are **range adaptors**. A view is like a window that enables you to "see" into a range and observe its elements—views do not have their own copy of those elements. Each range adaptor takes a `std::ranges::viewable_range` as an argument and returns a **view** of that range for use in a pipeline of operations (introduced in Section 6.14). A `viewable_range` is a range that can be "safely converted into a view."[36] Because views do not own the data that they "see," **a temporary object cannot be converted to a `viewable_range`**.

36. "`std::ranges::viewable_range`." Accessed January 29, 2022. https://en.cppreference.com/w/cpp/ranges/viewable_range.

20 The following table lists many **C++20 range adaptors** (header `<ranges>`),[37] which enable the **functional-style programming** techniques we introduced in Section 6.14. **Range adaptors** are defined in the namespace `std::ranges::views` and also can be accessed via the alias `std::views`.

37. "Ranges Library." Accessed January 29, 2022. https://en.cppreference.com/w/cpp/ranges.

Range adaptor	Description
`filter`	Creates a `view` representing only the **range** elements for which a predicate returns `true`.
`transform`	Creates a `view` that **maps** elements to new values.
`common`	Used to convert a `view` into a `std::ranges::common_range`, which enables a **range** to be used with **common-range algorithms** that require `begin/end` **iterator pairs** of the same type.
`all`	Creates a `view` representing all of a **range's** elements.
`counted`	Creates a `view` of a specified number of elements from either the beginning of a **range** or from a specific iterator **position**.

Range adaptor	Description
`reverse`	Creates a `view` for processing a **bidirectional** `view` in **reverse order**.
`drop`	Creates a `view` that ignores the specified number of elements at the beginning of another `view`.
`drop_while`	Creates a `view` that ignores the elements at the beginning of another `view` as long as a **predicate** returns `true`.
`take`	Creates a `view` containing the specified number of elements from the beginning of another `view`.
`take_while`	Creates a `view` containing the elements from the beginning of another `view` as long as a **predicate** returns `true`.
`join`	Creates a `view` that combines the elements of **multiple ranges**.
`split`	Splits a `view` based on a delimiter. The new `view` contains a separate `view` for each **subrange**.
`keys`	Creates a `view` of the **keys** in **key—value pairs**, such as those in a `map`. The **keys** are the first elements in the **pairs**.
`values`	Creates a `view` of the **values** in **key—value pairs**, such as those in a `map`. The **values** are the second elements in the **pairs**.
`elements`	For a `view` containing `tuple`s, `pair`s or `array`s, creates a `view` consisting of elements from a specified index in each object.

14.7.2 Working with Range Adaptors and Views

Figure 14.18[38] demonstrates several `std::views` from the preceding table and introduces the infinite version of `std::views::iota`. Line 13 defines a **lambda** that returns `true` if its argument is an even integer. We'll use this **lambda** in our **pipelines**.

38. As of January 2022, this program does not compile using the most recent `clang++`.

```
1  // fig14_18.cpp
2  // Working with C++20 std::views.
3  #include <algorithm>
4  #include <iostream>
5  #include <iterator>
6  #include <map>
7  #include <ranges>
```

```
 8  #include <string>
 9  #include <vector>
10
11  int main() {
12     std::ostream_iterator<int> output{std::cout, " "};
13     auto isEven{[](int x) {return x % 2 == 0;}}; // true if x is even
14
```

Fig. 14.18 Working with C++20 `std::views`.

Creating an Infinite Range with `std::views::iota`

SE In Fig. 6.13, we introduced **`std::views::iota`**, which is known as a **range factory**—it's a view that **lazily creates a sequence of consecutive integers** when you iterate over it. The version of **`iota`** in Fig. 6.13 required two arguments—the starting integer value and the value that's one past the end of the sequence that **`iota`** should produce. Line 16 uses **`iota`**'s **infinite range** version, which receives only the starting integer (`0`) in the sequence and increments it by one until you tell it to stop—you'll see how momentarily. The pipeline in line 16 creates a **view** that **`filter`**s the integers produced by **`iota`**, keeping only the integers for which the **lambda** **`isEven`** returns `true`. At this point, no integers have been produced. Again, **views are lazy**—they do not execute until you iterate through them with a loop or a standard library algorithm. **Views are objects that you can store in variables so you can reuse their processing steps and even add more processing steps later.** We store this **view** in `evens`, which we'll use `evens` to build several enhanced pipelines.

```
15     // infinite view of even integers starting at 0
16     auto evens{std::views::iota(0) | std::views::filter(isEven)};
17
```

`take` Range Adaptor

Err Though an **infinite range** is logically infinite,[39] to process one in a loop or pass one to a standard library algorithm, **you must limit the number of elements the pipeline will produce**; otherwise, your program will contain an infinite loop. One way to do this is to use a range adaptor that limits the number of elements to process. Such adaptors work with **infinite ranges** and **fixed-size ranges**. For example, line 19 uses the **`take` range adaptor** to take only the first five values from the **`evens` pipeline** we defined in line 16. We pass the resulting **view** to **`std::ranges::copy`** (line 19), which iterates through the **pipeline**, causing it to execute its steps:

39. Jeff Garland, "Using C++20 Ranges Effectively," June 18, 2019. Accessed January 29, 2022. https://www.youtube.com/watch?v=VmWS-9idT3s.

- **`iota`** produces an integer,
- **`filter`** checks if it's even and, if so,
- **`take`** passes that value to **`copy`**.

If the value **`iota`** produces is not even, **`filter`** discards that value and **`iota`** produces the next value in the sequence. This process repeats until **`take`** has passed the specified number of elements to **`copy`**. **You can take any number of items from an infinite range or up to the maximum number of items in a fixed-size range.** For demonstration purposes, we'll process just a few items in each of the subsequent pipelines we discuss.

```
18    std::cout << "First five even ints: ";
19    std::ranges::copy(evens | std::views::take(5), output);
20
```

```
First five even ints: 0 2 4 6 8
```

`take_while` Range Adaptor

Lines 22–23 create an enhanced pipeline that uses the **`take_while` range adaptor** to limit the `evens` infinite range. This range adaptor returns a view that takes elements from the earlier steps in the pipeline while `take_while`'s **unary predicate** returns `true`. In this case, we take even integers while those values are less than 12. The first value greater than or equal to 12 terminates the pipeline. We store the view returned by `take_while` in the variable `lessThan12` for use in subsequent statements. Line 24 passes `lessThan12` to **`std::ranges::copy`**, which iterates through the view—executing its pipeline steps—and displays the results.

```
21    std::cout << "\nEven ints less than 12: ";
22    auto lessThan12{
23       evens | std::views::take_while([](int x) {return x < 12;})};
24    std::ranges::copy(lessThan12, output);
25
```

```
Even ints less than 12: 0 2 4 6 8 10
```

`reverse` Range Adaptor

Line 27 uses the **`reverse` range adaptor** to reverse the integers from the view `lessThan12` from lines 22–23. We pass the view returned by **`reverse`** to **`std::ranges::copy`**, which iterates through the view—executing its pipeline steps—and displays the results.

```
26    std::cout << "\nEven ints less than 12 reversed: ";
27    std::ranges::copy(lessThan12 | std::views::reverse, output);
28
```

```
Even ints less than 12 reversed: 10 8 6 4 2 0
```

`transform` Range Adaptor

We introduced the **`transform` range adaptor** in Fig. 6.13. The pipeline in lines 31–33 creates a view that gets the integers produced by `lessThan12`, reverses them, then uses **`transform`** to square their values. We pass the resulting view to **`std::ranges::copy`**, which iterates through the view—executing its pipeline steps—and displays the results.

```
29    std::cout << "\nSquares of even ints less than 12 reversed: ";
30    std::ranges::copy(
31       lessThan12
32          | std::views::reverse
33          | std::views::transform([](int x) {return x * x;}),
34       output);
35
```

```
Squares of even ints less than 12 reversed: 100 64 36 16 4 0
```

`drop` Range Adaptor

The pipeline in line 38 begins with the **infinite sequence** of even integers produced by `evens`, uses the `drop` **range adaptor** to skip the first 1,000 even integers, then uses the `take` **range adaptor** to take the next five even integers in the sequence. We pass the resulting view to `std::ranges::copy`, which iterates through the view—executing its pipeline steps—and displays the results.

```
36    std::cout << "\nSkip 1000 even ints, then take five: ";
37    std::ranges::copy(
38       evens | std::views::drop(1000) | std::views::take(5),
39       output);
40
```

```
Skip 1000 even ints, then take five: 2000 2002 2004 2006 2008
```

`drop_while` Range Adaptor

You also can skip elements while a **unary predicate** remains true. The pipeline in lines 43–45 begins with the **infinite sequence** of even integers produced by `evens`. It uses the `drop_while` **range adaptor** to skip even integers at the beginning of the **infinite sequence** that are less than or equal to 1,000. Then it uses the `take` **range adaptor** to take the next five even integers in the sequence. We pass the resulting view to `std::ranges::copy`, which iterates through the view—executing its pipeline steps—and displays the results.

```
41    std::cout << "\nFirst five even ints greater than 1000: ";
42    std::ranges::copy(
43       evens
44          | std::views::drop_while([](int x) {return x <= 1000;})
45          | std::views::take(5),
46       output);
47
```

```
First five even ints greater than 1000: 1002 1004 1006 1008 1010
```

Creating and Displaying a `map` of Roman Numerals and Their Decimal Values

So far, we've focused on processing ranges of integers for simplicity, but you can process ranges of more complex types and process various containers as well. Next, we'll process the **key–value pairs** in a `map` containing `string` objects as keys and `int`s as values. The keys are roman numerals, and the values are their corresponding decimal values. The `using` declaration in line 49 enables us to use `string` **object literals** as we build each key–value pair in lines 51–52. For example, in the value `"I"s`, the `s` following the string literal designates that the literal is a `string` object. Lines 53–54 create a lambda that we use in line 56 with `std::ranges::for_each` to display each key–value pair in the `map`.

```
48    // allow std::string object literals
49    using namespace std::string_literals;
50
51    std::map<std::string, int> romanNumerals{
52       {"I"s, 1}, {"II"s, 2}, {"III"s, 3}, {"IV"s, 4}, {"V"s, 5}};
53    auto displayPair{[](const auto& p) {
54          std::cout << p.first << " = " << p.second << "\n";}};
```

```
55    std::cout << "\n\nromanNumerals:\n";
56    std::ranges::for_each(romanNumerals, displayPair);
57
```

```
romanNumerals:
I = 1
II = 2
III = 3
IV = 4
V = 5
```

`keys` and `values` Range Adaptors

When working with `map`s in pipelines, each key–value pair is treated as a `pair` object containing a key and a value. You can use the range adaptors `keys` (line 60) and `values` (line 63) to get **views of only the keys and values**, respectively:

- `keys` creates a **view** that selects only the *first* item in each `pair`.
- `values` creates a **view** that selects only the *second* item in each `pair`.

We pass each pipeline to `std::ranges::copy`, which iterates through the pipeline and displays the results.

```
58    std::ostream_iterator<std::string> stringOutput{std::cout, " "};
59    std::cout << "\nKeys in romanNumerals: ";
60    std::ranges::copy(romanNumerals | std::views::keys, stringOutput);
61
62    std::cout << "\nValues in romanNumerals: ";
63    std::ranges::copy(romanNumerals | std::views::values, output);
64
```

```
Keys in romanNumerals: I II III IV V
Values in romanNumerals: 1 2 3 4 5
```

`elements` Range Adaptor

Interestingly, the `keys` and `values` **range adaptors** also work with ranges in which each element is a `tuple` or an `array`. Even if they contain more than two elements each, `keys` **always selects the first item**, and `values` **always selects the second**. If the `tuple` or `array` elements in the range have more than two elements, the `elements` **range adaptor can select items by index**, as we demonstrate with `romanNumerals`' **key–value pairs**. Line 67 selects only element `0` from each `pair` object in the range, and line 70 selects only element `1`. In both cases, we pass the pipeline to `std::ranges::copy`, which iterates through the pipelines and displays the results.

```
65    std::cout << "\nKeys in romanNumerals via std::views::elements: ";
66    std::ranges::copy(
67       romanNumerals | std::views::elements<0>, stringOutput);
68
69    std::cout << "\nvalues in romanNumerals via std::views::elements: ";
70    std::ranges::copy(romanNumerals | std::views::elements<1>, output);
71    std::cout << "\n";
72 }
```

```
Keys in romanNumerals via std::views::elements: I II III IV V
values in romanNumerals via std::views::elements: 1 2 3 4 5
```

Perf For decades, every couple of years computer processing power approximately doubled inexpensively. This is known as **Moore's law**—named for Gordon Moore, co-founder of Intel and the person who identified this trend in the 1960s. Key executives at computer-processor companies NVIDIA and Arm have indicated that Moore's law no longer applies.[40,41] Computer processing power continues to increase, but hardware vendors now rely on new processor designs, such as **multi-core processors**, which enable true parallel processing for better performance. C++ has always been focused on performance. However, its first standard support for parallelism was not added until C++11—32 years after the language's inception.

40. Esther Shein, "Moore's Law Turns 55: Is It Still Relevant?" April 17, 2020. Accessed January 29, 2022. https://www.techrepublic.com/article/moores-law-turns-55-is-it-still-relevant.
41. Nick Heath, "Moore's Law Is Dead: Three Predictions About the Computers of Tomorrow," September 19, 2018. Accessed January 29, 2022. https://www.techrepublic.com/article/mooreslaw-is-dead-three-predictions-about-the-computers-of-tomorrow/.

Parallelizing an algorithm is not as simple as "flipping a switch" to say, "I want to run this algorithm in parallel." Parallelization is sensitive to what the algorithm does and which parts can truly run in parallel on multicore hardware. Programmers must carefully determine how to divide tasks for parallel execution. Such algorithms must be scalable to any number of cores—more or fewer cores might be available at a given time because they're shared among all the computer's tasks. In addition, the number of cores is increasing over time as computer architecture evolves, so algorithms should be flexible enough to take advantage of those additional cores. As challenging as it is to write parallel algorithms, the incentive is high to maximize application performance.

17 Over the years, it has become clear that designing algorithms capable of executing on multiple cores is complex and error-prone. Many programming languages now provide built-in library capabilities that offer "canned" parallelism features. C++ already has a collection of valuable algorithms. To help programmers avoid "reinventing the wheel," C++17 introduced **parallel overloads** for **69 common-ranges algorithms**, enabling them to take advantage of multicore architectures and the high-performance "vector mathematics" operations available on today's CPUs and GPUs. Vector operations can perform the same operation on many data items simultaneously.[42]

42. "General Utilities Library—execution Policies—unsequenced Execution Policy." Accessed January 29, 2022. https://timsong-cpp.github.io/cppwp/n4861/execpol.unseq.

17 In addition, C++17 added seven new parallel algorithms, each of which also has a sequential version:

- `for_each_n`
- `exclusive_scan`
- `inclusive_scan`
- `transform_exclusive_scan`
- `transform_inclusive_scan`
- `reduce`
- `transform_reduce`

23 The algorithm overloads take the burden of parallel programming largely off programmers' shoulders and place it on the prepackaged library algorithms, leveraging the programming process.

Unfortunately, the **C++20 `std::ranges` algorithms** are not yet parallelized, though they might be for C++23.[43]

43. Barry Revzin, Conor Hoekstra and Tim Song, "A Plan for C++23 Ranges," October 14, 2020. Accessed January 29, 2022. https://wg21.link/p2214r0.

Chapter 17, Parallel Algorithms and Concurrency: A High-Level View, will

- overview the parallel algorithms,
- 17 20 discuss the four "execution policies" (three from C++17 and one from C++20) that determine how a parallel algorithm uses a system's parallel processing capabilities to perform a task,
- demonstrate how to invoke parallel algorithms and
- use functions from the `<chrono>` header to time how long it takes to execute standard library algorithms running sequentially vs. running in parallel so you can see the difference in performance.

20 You'll see that the parallel versions of algorithms do not always run faster than sequential versions—and we'll explain why. We'll also introduce the various standard library headers containing C++'s concurrency and parallel-programming capabilities. Then, in Chapter 18, we'll introduce C++20's new coroutines feature.

14.9 Standard Library Algorithm Summary

The C++ standard specifies 117 algorithms—many overloaded with two or more versions. The standard separates the algorithms into several categories:

- mutating sequence algorithms (`<algorithm>`),
- nonmodifying sequence algorithms (`<algorithm>`),
- sorting and related algorithms (`<algorithm>`),
- generalized numeric operations (`<numeric>`) and
- specialized memory operations (`<memory>`).

To learn about algorithms we did not present in this chapter, visit sites such as

```
https://en.cppreference.com/w/cpp/algorithm
https://docs.microsoft.com/en-us/cpp/standard-library/algorithm
```

Throughout this section's tables:

- Algorithms we present in this chapter are grouped at the top of each table and shown in **`bold`**.
- Algorithms that have a **C++20 `std::ranges` overload** are marked with a superscript "R."[44]

44. "Constrained Algorithms." Accessed January 29, 2022. https://en.cppreference.com/w/cpp/algorithm/ranges.

- Algorithms that have a parallel version are marked with a superscript "P."[45]

45. "Extensions for Parallelism." Accessed January 29, 2022. https://en.cppreference.com/w/cpp/experimental/parallelism/. [Though this link contains "experimental," these parallel versions are officially part of the C++ standard.]

- Algorithms added in C++ versions 11, 17 and 20 are marked with the superscript version number.

For example, the algorithm **`copy`** is marked with "PR," meaning it has a parallelized version and a `std::ranges` version, and the algorithm **`is_sorted_until`** is marked with "PR11," meaning it has a parallelized version and a `std::ranges` version and it was introduced to the standard library in C++11.

Mutating Sequence Algorithms

Sec The following table shows many of the **mutating-sequence algorithms**—that is, algorithms that modify the containers on which they operate. The `shuffle` algorithm replaced the less-secure `random_shuffle` algorithm. "Under the hood," `random_shuffle` used function `rand`—which was inherited into C++ from the C standard library. C's `rand` does not have "good statistical properties" and can be predictable,[46] making programs that use `rand` less secure. The newer **`shuffle`** algorithm uses the **C++11 nondeterministic random-number generation** capabilities.

46. "Do Not Use the `rand()` Function for Generating Pseudorandom Numbers." Last modified April 23, 2021. Accessed January 29, 2022. https://wiki.sei.cmu.edu/confluence/display/c/MSC30-C.+Do+not+use+the+rand%28%29+function+for+generating+pseudorandom+numbers.

Mutating sequence algorithms from header `<algorithm>`

`copy`[PR]	**`copy_backward`**[R]	**`copy_if`**[PR11]	**`copy_n`**[PR11]
`fill`[PR]	**`fill_n`**[PR]	**`generate`**[PR]	**`generate_n`**[PR]
`iter_swap`	**`remove`**[PR]	**`remove_copy`**[PR]	**`remove_copy_if`**[PR]
`remove_if`[PR]	**`replace`**[PR]	**`replace_copy`**[PR]	**`replace_copy_if`**[PR]
`replace_if`[PR]	**`reverse`**[PR]	**`reverse_copy`**[PR]	**`shuffle`**[R11]
`swap_ranges`[PR]	**`transform`**[PR]	**`unique`**[PR]	**`unique_copy`**[PR]
`move`[PR11]	`move_backward`[R11]	`rotate`[PR]	**`rotate_copy`**[PR]
`sample`[R17]	`shift_left`[20]	`shift_right`[20]	

Nonmodifying Sequence Algorithms

The following table shows the **nonmodifying sequence algorithms**—that is, algorithms that do not modify the containers they manipulate.

Nonmodifying sequence algorithms from header `<algorithm>`

`all_of`[PR11]	**`any_of`**[PR11]	**`count`**[PR]	**`count_if`**[PR]
`equal`[PR]	**`find`**[PR]	**`find_if`**[PR]	**`find_if_not`**[PR11]
`for_each`[PR]	**`mismatch`**[PR]	**`none_of`**[PR11]	
`adjacent_find`[PR]	`find_end`[PR]	`find_first_of`[PR]	`for_each_n`[PR17]
`is_permutation`[R11]	`search`[PR]	`search_n`[PR]	

Sorting and Related Algorithms

The following table shows the sorting and related algorithms.

Sorting and related algorithms from header `<algorithm>`		
binary_search[R]	equal_range[R]	includes[PR]
inplace_merge[PR]	lexicographical_compare[PR]	lower_bound[R]
make_heap[R]	max[R]	max_element[PR]
merge[PR]	min[R]	min_element[PR]
minmax[R11]	minmax_element[PR11]	pop_heap[R]
push_heap[R]	set_difference[PR]	set_intersection[PR]
set_symmetric_difference[PR]	set_union[PR]	sort[PR]
sort_heap[R]	upper_bound[R]	
clamp[R17]	is_heap[PR11]	is_heap_until[PR11]
is_partitioned[PR11]	is_sorted[PR11]	is_sorted_until[PR11]
lexicographical_compare_three_way[20]		next_permutation[R]
nth_element[PR]	partial_sort[PR]	partial_sort_copy[PR]
partition[PR]	partition_copy[PR11]	partition_point[R11]
prev_permutation[R]	stable_partition[PR]	stable_sort[PR]

Numerical Algorithms

23 The following table shows the numerical algorithms of the header `<numeric>`. The algorithms in this header have not yet been updated for C++20 ranges, though they are being worked on for inclusion in C++23.[47,48]

47. Christopher Di Bella, "A Concept Design for the Numeric," August 2, 2019. Accessed January 29, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1813r0.pdf.

48. Tristan Brindle, "Numeric Range Algorithms for C++20," May 19, 2020. Accessed January 29, 2022. https://tristanbrindle.com/posts/numeric-ranges-for-cpp20.

Generalized numeric operations from header `<numeric>`		
accumulate	gcd[17]	iota[11]

Generalized numeric operations from header `<numeric>`		
`lcm`[17]	`partia1_sum`	`reduce`[P17]
`adjacent_difference`[P]	`exclusive_scan`[P17]	`inclusive_scan`[P17]
`inner_product`[P]	`midpoint`[20]	`transform_reduce`[P17]
`transform_exclusive_scan`[P17]	`transform_inclusive_scan`[P17]	

Specialized Memory Operations

The following table shows the specialized memory algorithms of the header `<memory>`, which contains features for dynamic memory manipulation operations that are beyond this book's scope. For an overview of the header and these algorithms, see

```
https://en.cppreference.com/w/cpp/header/memory
```

Specialized Memory Operations	
`construct_at`[R20]	`destroy`[R17]
`destroy_at`[R17]	`destroy_n`[R17]
`uninitialized_copy`[PR]	`uninitialized_copy_n`[PR11]
`uninitialized_default_construct`[PR17]	`uninitialized_default_construct_n`[PR17]
`uninitialized_fill`[PR]	`uninitialized_fill_n`[PR]
`uninitialized_move`[PR17]	`uninitialized_move_n`[PR17]
`uninitialized_value_construct`[PR17]	`uninitialized_value_construct_n`[PR17]

14.10 A Look Ahead to C++23 Ranges

23 Though many algorithms have overloads in the **C++20 `std::ranges` namespace**, various C++20 algorithms—including those in the `<numeric>` header and the parallel algorithms in the `<algorithm>` header—do not yet have C++20 `std::ranges` overloads. There are standard committee proposals for various additional ranges library features that might be part of C++23. Some of these capabilities are under development and can be used now via the open-source project "Ranges for C++23."[49] C++20's ranges functionality and many of the new features proposed for C++23 are based on capabilities found in the open-source project range-v3.[50]

49. Corentin Jabot, “Ranges for C++23.” Accessed January 29, 2022. https://github.com/cor3ntin/rangesnext.
50. Eric Niebler, “range-v3.” Accessed January 29, 2022. https://github.com/ericniebler/range-v3.

“A Plan for C++23 Ranges”[51]

51. Barry Revzin, Conor Hoekstra and Tim Song, “A Plan for C++23 Ranges,” October 14, 2020. Accessed January 29, 2022. https://wg21.link/p2214r0.

This paper provides an overview of the general plan for **C++23 ranges**, plus details on many possible new ranges features. The proposed features are categorized by importance into three tiers, with Tier 1 containing the most important features. After a brief introduction, the paper discusses several categories of possible additions:

- **View adjuncts**: This section briefly discusses two key features—the **overloaded `ranges::to` function** for converting views to various container types and the ability to format views and ranges for convenient output. These are discussed in detail in the papers “`ranges::to`: A Function to Convert Any Range to a Container”[52] and “Formatting Ranges,”[53] respectively.

52. Corentin Jabot, Eric Niebler and Casey Carter, “`ranges::to`: A Function to Convert Any Range to a Container,” November 22, 2021. Accessed January 29, 2022. https://wg21.link/p1206r3.
53. Barry Revzin, “Formatting Ranges,” February 19, 2021. Accessed January 29, 2022. https://wg21.link/p2286.

- **Algorithms**: This section overviews potential new **`std::ranges` overloads** for various **`<numeric>` algorithms**, which are discussed in more detail in the paper “A Concept Design for the Numeric Algorithms.”[54] This section also briefly overviews potential issues with producing **`std::ranges` overloads** of C++17’s parallel algorithms. The paper “Introduce Parallelism to the Ranges TS”[55] provides more in-depth discussions of these issues.

54. Christopher Di Bella, “A Concept Design for the Numeric Algorithms,” August 2, 2019. Accessed January 29, 2022. http://wg21.link/p1813r0.
55. Gordon Brown, Christopher Di Bella, Michael Haidl, Toomas Remmelg, Ruyman Reyes, Michel Steuwer and Michael Wong, “P0836R1 Introduce Parallelism to the Ranges TS,” May 7, 2018. Accessed January 29, 2022. https://wg21.link/p0836r1.

- **Actions**: These are a third category of capabilities separate from ranges and views in the **range-v3 project**. Actions, like **views**, are composable with the **`|` operator**, but, like the **`std::ranges` algorithms**, actions are **greedy**, so they immediately produce results. According to this section, though actions would make some coding more convenient, adding them is a low priority because you can perform the same tasks by calling existing `std::ranges` algorithms in multiple statements.

14.11 Wrap-Up

In this chapter, we demonstrated many of the standard library algorithms, including filling containers with values, generating values, comparing elements or entire containers, removing elements, replacing elements, mathematical operations, searching, sorting, swapping, copying, merging, set operations, determining boundaries, and calculating minimums and maximums. We focused primarily on the C++20 `std::ranges` versions of these algorithms.

You saw that the standard library’s algorithms specify various minimum requirements that help you determine which containers, iterators and functions can be passed to each algorithm. We overviewed some named requirements used by the common-ranges algorithms, then indicated that the C++20 range-based algorithms use C++20 concepts to specify their requirements, which are

checked at compile-time. We briefly introduced the C++20 concepts specified for each range-based algorithm we presented. Not all algorithms have a range-based version.

We revisited lambdas and introduced additional capabilities for capturing the enclosing scope's variables. You saw that many algorithms can receive a lambda, a function pointer or a function object as an argument, and call them to customize the algorithms' behaviors.

We continued our discussion of C++'s functional-style programming. We showed how to create a logically infinite sequence of values and how to use range adaptors to limit the total number of elements processed through a pipeline. We saved views in variables for later use and added more steps to a previously saved pipeline. We introduced range adaptors for manipulating the keys and values in key–value pairs, and showed a similar range adaptor for selecting any indexed element from fixed-size objects, like `pair`s, `tuple`s and `array`s.

You learned that C++17 introduced new parallel overloads for 69 standard library algorithms in the `<algorithm>` header. As you'll see in Chapter 17, these will enable you to take advantage of your computer's multi-core hardware to enhance program performance. In Chapter 17, we'll demonstrate several parallel algorithms and use the `<chrono>` header's capabilities to time sequential and parallel algorithm calls so you can see the performance differences. We'll explain why parallel algorithms do not always run faster than their sequential counterparts, so it's not always worthwhile to use the parallel versions.

You saw that various C++20 algorithms, including those in the `<numeric>` header and the parallel algorithms in the `<algorithm>` header, do not have `std::ranges` overloads. We mentioned the updates expected in C++23 and pointed you to the GitHub project `rangesnext`, which contains implementations for many proposed updates.

In the next chapter, we'll build a simple container, iterators and a simple algorithm using custom templates, using C++20 concepts in our templates as appropriate.

15. Templates, C++20 Concepts and Metaprogramming

Objectives

In this chapter, you'll:

- Appreciate the rising importance of generic programming.
- Use class templates to create related custom classes.
- Understand compile-time vs. runtime polymorphism.
- Distinguish between templates and template instantiations.
- Use C++20 abbreviated function templates and templated lambdas.
- Use C++20 concepts to constrain template parameters and overload function templates based on their type requirements.
- Use type traits and see how they relate to C++20 concepts.
- Test concepts at compile-time with `static_assert`.
- Create a custom concept-constrained algorithm.
- Rebuild our class `MyArray` as a custom container class template with custom iterators.
- Use non-type template parameters to pass compile-time constants to templates and use default template arguments.
- Use variadic templates that receive any number of parameters and apply binary operators to them via fold expressions.
- Use compile-time template metaprogramming capabilities to compute values, manipulate types and generate code to improve runtime performance.

Outline

15.1 Introduction
15.2 Custom Class Templates and Compile-Time Polymorphism
15.3 C++20 Function Template Enhancements
15.3.1 C++20 Abbreviated Function Templates
15.3.2 C++20 Templated Lambdas
15.4 C++20 Concepts: A First Look
15.4.1 Unconstrained Function Template `multiply`
15.4.2 Constrained Function Template with a C++20 Concepts `requires` Clause
15.4.3 C++20 Predefined Concepts
15.5 Type Traits
15.6 C++20 Concepts: A Deeper Look
15.6.1 Creating a Custom Concept
15.6.2 Using a Concept
15.6.3 Using Concepts in Abbreviated Function Templates
15.6.4 Concept-Based Overloading
15.6.5 `requires` Expressions
15.6.6 C++20 Exposition-Only Concepts
15.6.7 Techniques Before C++20 Concepts: SFINAE and Tag Dispatch
15.7 Testing C++20 Concepts with `static_assert`
15.8 Creating a Custom Algorithm
15.9 Creating a Custom Container and Iterators
15.9.1 Class Template `ConstIterator`

15.9.2 Class Template `Iterator`
15.9.3 Class Template `MyArray`
15.9.4 MyArray Deduction Guide for Braced Initialization
15.9.5 Using `MyArray` and Its Custom Iterators with `std::ranges` Algorithms
15.10 Default Arguments for Template Type Parameters
15.11 Variable Templates
15.12 Variadic Templates and Fold Expressions
15.12.1 `tuple` Variadic Class Template
15.12.2 Variadic Function Templates and an Intro to C++17 Fold Expressions
15.12.3 Types of Fold Expressions
15.12.4 How Unary Fold Expressions Apply Their Operators
15.12.5 How Binary-Fold Expressions Apply Their Operators
15.12.6 Using the Comma Operator to Repeatedly Perform an Operation
15.12.7 Constraining Parameter Pack Elements to the Same Type
15.13 Template Metaprogramming
15.13.1 C++ Templates Are Turing Complete
15.13.2 Computing Values at Compile-Time
15.13.3 Conditional Compilation with Template Metaprogramming and `constexpr if`
15.13.4 Type Metafunctions
15.14 Wrap-Up

15.1 Introduction

CG Generic programming with templates has been in C++ since the 1998 C++ standard was released[1] and has increased in importance with each new release. A modern C++ theme is to do more at compile-time for better type checking and better runtime performance.[2,3,4] You've already used templates extensively. As you'll see, templates and template metaprogramming are the keys to powerful compile-time operations. In this chapter, we'll take a deeper look at templates as we explain how to **develop custom class templates**, explore **concepts**—C++20's most significant new feature—and introduce **template metaprogramming**. The following table summarizes the book's templates coverage across all 20 chapters.

---

1. "History of C++." Accessed February 2, 2022. https://www.cplusplus.com/info/history/.
2. C++ Core Guidelines, "Per: Performance." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-performance.
3. "Big Picture Issues—What's the Big Deal with Generic Programming?" Accessed February 2, 2022. https://isocpp.org/wiki/faq/big-picture#generic-paradigm.
4. "Compile Time vs. Run Time Polymorphism in C++: Advantages/Disadvantages." Accessed February 2, 2022. https://stackoverflow.com/questions/16875989/compile-time-vs-run-time-polymorphism-in-c-advantages-disadvantages.

Chapter	Templates coverage
Chapter 1	Introduction to **generic programming**.

Chapter	Templates coverage
Chapters 2–4	**strings** (which are **class templates**—`string` is an alias for `basic_string`).
Chapter 5	Section 5.8: `uniform_int_distribution` **class template** for random-number generation. Section 5.16: Defining a **function template**.
Chapter 6	**Standard library** container class templates `array` and `vector`.
Chapter 7	Section 7.10: **Class template span** for creating a view into a container of contiguous elements, such as an `array` or `vector`.
Chapter 8	Sections 8.2–8.9: In-depth treatment of `strings` (which are **class templates**). Sections 8.12–8.16: **File-stream-processing classes** (which are **class templates**). Section 8.17: `string`-**stream class templates**. Section 8.19: `rapidcsv` **library function templates for manipulating CSV data**.
Chapter 9	Section 9.22: `cereal` **library function templates** for **serializing and deserializing data** using **JavaScript Object Notation (JSON)**.
Chapter 10	Section 10.13: **Runtime polymorphism** with the `std::variant` **class template** and the `std::visit` **function template**.
Chapter 11	**Smart pointers** for managing dynamically allocated memory with **class template** `unique_ptr`, **function template** `make_unique` and **class template** `initializer_list` for passing initializer lists to functions.
Chapter 12	Section 12.8: `unique_ptr` **class template.**
Chapter 13	Standard library **container class templates** and **iterators** (which also are implemented as **class templates**).
Chapter 14	Standard library **algorithm function templates** that manipulate standard library **container class templates** via **iterators** .
Chapter 15	**Custom class templates, iterator templates, function templates, abbreviated function templates, templated lambdas, type traits, C++20 concepts, concept-based overloading, variable**

Chapter	Templates coverage
	templates, **alias templates**, **variadic templates**, **fold expressions** and **template metaprogramming**.
Chapter 16	Parallel standard library algorithms and various other **function templates** and **class templates** related to multithreaded application development.
Chapter 19	**(Online) Stream I/O classes** (which are **class templates**).
Chapter 20	**(Online)** `shared_ptr` and `weak_ptr` smart-pointer **class templates**.

Custom Templates

Section 5.16 showed that the compiler uses a function template to generate overloaded functions—known as **instantiating the function template**. Similarly, the compiler uses class templates to generate related classes—this is known as **instantiating the class template**. Instantiating templates enables **compile-time (static) polymorphism**. In this chapter, you'll create custom class templates and study key template-related technologies.

C++20 Template Features

20 Concepts We discuss C++20's new template capabilities, including **abbreviated function templates**, **templated lambdas** and **concepts**—a C++20 "big four" feature that makes generic programming with templates even more convenient and powerful. When invoking C++20's **range-based algorithms**, you saw that you must pass container or iterator arguments that meet the algorithms' requirements. This chapter takes a template developer's viewpoint as we develop custom templates that **specify their requirements** using predefined and custom C++20 concepts.[5,6,7,8] The compiler checks concepts *before* instantiating templates' bodies, often resulting in fewer, clearer and more-precise error messages.

5. Marius Bancila, "Concepts Versus SFINAE-Based Constraints." Accessed February 2, 2022. https://mariusbancila.ro/blog/2019/10/04/concepts-versus-sfinae-based-constraints/.
6. Saar Raz, "C++20 Concepts: A Day in the Life," YouTube video, October 17, 2019. https://www.youtube.com/watch?v=qawSiMIXtE4.
7. "Constraints and Concepts." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/constraints.
8. "Concepts Library." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/concepts.

Type Traits

We'll introduce **type traits** for testing attributes of built-in and custom class types at compile-time. You'll see that many C++20 concepts are implemented in terms of type traits. Concepts simplify using type traits to constrain template parameters.

Building Custom Containers, Iterators and Algorithms

Chapter 11's `MyArray` class stored only `int` values. We'll create a `MyArray` class template that can be specialized to store elements of various types (e.g., `MyArray` of `float` or `MyArray` of `Employee`). We'll make `MyArray` objects compatible with many standard library algorithms by defining **custom iterators**. We'll also define a **custom algorithm** that can process `MyArray` elements and standard library container class objects.

Variadic Templates and Fold Expressions

17 We'll build a **variadic function template** that receives a variable number of parameters. We'll also introduce **C++17 fold expressions**, which enable you to conveniently apply an operation to all the items passed to a variadic template.

Template Metaprogramming

Perf The C++ Core Guidelines define template metaprogramming (TMP) as **"creating programs that compose code at compile time."**[9] The compiler also can use them to perform **compile-time calculations** and **type manipulations**. Compile-time calculations enable you to improve a program's execution-time performance, possibly reducing execution time and memory consumption. You'll see that **type traits** are used extensively in template metaprogramming for generating code based on template-argument attributes. We'll also write a function template that generates different code at compile-time, based on whether its container argument supports **random-access iterators** or **lesser iterators**. You'll see how this enables us to **optimize the program's runtime performance**.

---

9. C++ Core Guidelines, "T: Templates and Generic Programming." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-templates.

20 CG Before C++20, many C++ programmers viewed template metaprogramming as too complex to use. C++20 concepts make aspects of it friendlier and more accessible. Nevertheless, Google's C++ Style Guide says to "avoid complicated template programming."[10] Similarly, the C++ Core Guidelines indicate that you should "use template metaprogramming only when you really need to" and that it's "hard to get right, ... and is often very hard to maintain."[11] These guidelines could be updated when developers gain more experience using C++20 concepts and enhancements possibly coming in C++23.

10. "Google C++ Style Guide—Template Metaprogramming." Accessed February 2, 2022. https://google.github.io/styleguide/cppguide.html#Template_metaprogramming.
11. C++ Core Guidelines, "T.120: Use Template Metaprogramming Only When You Really Need To." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-metameta.

15.2 Custom Class Templates and Compile-Time Polymorphism

A template enables **compile-time** (or **static**) **polymorphism**[12,13] by specifying capabilities generically, then letting the compiler instantiate the template, generating type-specific code specializations on demand. Class templates are called **parameterized types** because they require one or more **parameters** to tell the compiler how to customize a class template to form a **class-template specialization** from which objects can be instantiated. You write one class-template definition. **When you need particular specializations, you use concise, simple notations to instantiate the template, and**

the compiler writes only those specializations for you. One `Stack` class template, for example, could become the basis for creating many `Stack` class-template specializations, such as "`Stack` of `double`s," "`Stack` of `int`s," "`Stack` of `Employee`s," "`Stack` of `Bill`s" or "`Stack` of `ActivationRecord`s."
12. "C++—Static Polymorphism." Wikipedia. Wikimedia Foundation. Accessed February 2, 2022. https://en.wikipedia.org/wiki/C%2B%2B#Static_polymorphism.
13. Kateryna Bondarenko, "Static Polymorphism in C++," May 6, 2019. Accessed February 2, 2022. https://medium.com/@kateolenya/static-polymorphism-in-c-9e1ae27a945b.

To use a type with a template, the type must meet the template's requirements. For example, a template might require objects of a specified type to

- be **initializable with a default constructor**,
- be **copyable** or **movable**,
- be **comparable with operator `<`** to determine their sort order,
- have **specific member functions** and more.

Err **Compilation errors usually occur when you instantiate a template with a type that does not meet the template's requirements.**

Creating Class Template `Stack<T>`

SE Let's jump into coding a custom `Stack` class template. It's possible to understand the notion of a stack **independently of the kinds of items you place onto or remove from it**—stacks are simply **last-in, first-out (LIFO) data structures. Class templates encourage software reuse by enabling the compiler to instantiate many type-specific class-template specializations from a single class template**—as you saw with the **`stack` container adapter** in Section 13.10.1. Here (Fig. 15.1), we define a stack generically, then instantiate and use type-specific stacks (Fig. 15.2). Figure 15.1's `Stack` class-template definition looks like a conventional class definition, with a few key differences.

```
1  // Fig. 15.1: Stack.h
2  // Stack class template.
3  #pragma once
4  #include <deque>
5
6  template<typename T>
7  class Stack {
8  public:
9     // return the top element of Stack
10    const T& top() const {return stack.front();}
11
12    // push an element onto Stack
13    void push(const T& pushValue) {stack.push_front(pushValue);}
14
15    // pop an element from Stack
16    void pop() {stack.pop_front();}
17
18    // determine whether Stack is empty
19    bool isEmpty() const {return stack.empty();}
20
21    // return size of Stack
22    size_t size() const {return stack.empty();}
```

```
23   private:
24      std::deque<T> stack{}; // internal representation of Stack
25   };
```

Fig. 15.1 `Stack` class template.

`template` Header

The first key difference is the **`template` header** (line 6)

```
template<typename T>
```

which begins with the **`template`** keyword, followed by a comma-separated list of **template parameters** enclosed in **angle brackets** (< and >). Each template parameter representing a type must be preceded by one of the interchangeable keywords **`typename`** or `class`. Some programmers prefer `typename` because a template's **type arguments** might not be class types. (There are other cases in which `typename` must be used rather than `class`.[14,15]) The **type parameter** `T` is a **placeholder** for the **`Stack`'s element type**. Type parameter names can be any valid identifier but must be unique inside a template definition. `T` is mentioned throughout the `Stack` class template wherever we need access to the `Stack`'s element type:

14. Scott Meyers, "Item 42: Understand the Meanings of `typename`," *Effective C++, 3/e*. Pearson Education, Inc., 2005.
15. John Lakos, "1.3 Declarations, Definitions, and Linkage," *Large-Scale C++.* Addison-Wesley, 2020. Footnotes 56 and 57.

- declaring a member function return type (line 10),
- declaring a member function parameter (line 13) and
- declaring a variable (line 24).

The compiler associates the type parameter with a **type argument** when you instantiate the class template. At that point, the compiler generates a copy of the class template in which all occurrences of the type parameter are replaced with the specified type. **Compilers generate definitions only for the portions of a template used in your code.**[16,17]

16. Andreas Fertig, "Back to Basics: Templates (Part 1 of 2)," September 25, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=VNJ4wiuxJM4.
17. "13.9.2 Implicit Instantiation [temp.inst]." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/temp.inst#4.

SE Another difference between class template `Stack` and other classes we've defined is that **we did not separate the class template's interface from its implementation**. You define templates in headers, then `#include` them in client-code files. **The compiler needs the full template definition each time the template is instantiated with new type arguments to generate the appropriate code.** For class templates, this means that the member functions also are defined in the header—typically inside the class definition's body, as in Fig. 15.1.

Class Template `Stack<T>`'s Data Representation

Perf Section 13.10.1 showed that the standard library's **`stack` adapter class** can use various containers to store its elements. A `stack` requires insertions and deletions only at its **top**, so a **`vector`** or a **`deque`** could be used to store the `stack`'s elements. A `vector` supports fast insertions and deletions at its *back*. A `deque` supports fast insertions and deletions at its *front* and its *back*. A `deque` is the default

representation for the standard library's `stack` adapter[18] because a **`deque` grows more efficiently than a `vector`**

18. "`std::stack`." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/container/stack.

- A **`vector`**'s elements are stored in a contiguous block of memory. When that block is full and you add a new element, the `vector` performs the **expensive operation of allocating a larger contiguous block of memory** and **copying** or **moving** the old elements into that new block.
- A **`deque`** is typically implemented as a list of fixed-size, built-in arrays—with new ones added as necessary. **No existing elements are copied or moved when new items are added to a `deque`'s front or back**.

For these reasons, we use a `deque` (line 24) as our `Stack` class's underlying container.

Class Template `Stack<T>`'s Member Functions

A class template's member-function definitions behave like function templates. **When you define them within the class template's body, you do not precede them with a `template` header**. They still use the template parameter `T` to represent the element type. The `Stack` class template does not define its own constructors—**the class's default constructor invokes the `deque` data member's default constructor**.

`Stack` (Fig. 15.1) provides the following member functions:

- **`top`** (line 10) returns a `const` reference to the `Stack`'s top element without removing it. You could overload `top` with a non-`const` version as well.
- **`push`** (line 13) places a new element on the top of the `Stack`.
- **`pop`** (line 16) removes the `Stack`'s top element.
- **`isEmpty`** (line 19) returns the `bool` value `true` if the `Stack` is empty; otherwise, it returns `false`.
- **`size`** (line 22) returns the number of elements in the `Stack`.

Each calls a **`deque`** member function to perform its task—this is known as **delegation**.

Testing Class Template `Stack<T>`

Figure 15.2 tests the **`Stack` class template**. Line 8 instantiates `doubleStack`. This variable is declared as type **`Stack<double>`**—pronounced "`Stack` of `double`." The compiler associates type `double` with **type parameter `T`** in the class template to produce the source code for a `Stack` class with `double` elements that stores its elements in a **`deque<double>`**. Lines 15–19 invoke `push` (line 16) to place the `double` values 1.1, 2.2, 3.3, 4.4 and 5.5 onto `doubleStack`. Next, lines 24–27 invoke `isEmpty`, `top` and `pop` in a `while` loop to remove the stack's elements. The output shows that the values indeed `pop` off in **last-in, first-out order**. When `doubleStack` is empty, the `pop` loop terminates.

```
1  // fig15_02.cpp
2  // Stack class template test program.
3  #include <iostream>
4  #include "Stack.h" // Stack class template definition
5  using namespace std;
6
7  int main() {
8     Stack<double> doubleStack{}; // create a Stack of double
9     constexpr size_t doubleStackSize{5}; // stack size
10    double doubleValue{1.1}; // first value to push
11
12    cout << "Pushing elements onto doubleStack\n";
13
```

```
14      // push 5 doubles onto doubleStack
15      for (size_t i{0}; i < doubleStackSize; ++i) {
16          doubleStack.push(doubleValue);
17          cout << doubleValue << ' ';
18          doubleValue += 1.1;
19      }
20
21      cout << "\n\nPopping elements from doubleStack\n";
22
23      // pop elements from doubleStack
24      while (!doubleStack.isEmpty()) { // loop while Stack is not empty
25         cout << doubleStack.top() << ' '; // display top element
26         doubleStack.pop(); // remove top element
27      }
28
29      cout << "\nStack is empty, cannot pop.\n";
30
31      Stack<int> intStack{}; // create a Stack of int
32      constexpr size_t intStackSize{10}; // stack size
33      int intValue{1}; // first value to push
34
35      cout << "\nPushing elements onto intStack\n";
36
37     // push 10 integers onto intStack
38     for (size_t i{0}; i < intStackSize; ++i) {
39         intStack.push(intValue);
40         cout << intValue++ << ' ';
41      }
42
43      cout << "\n\nPopping elements from intStack\n";
44
45      // pop elements from intStack
46      while (!intStack.isEmpty()) { // loop while Stack is not empty
47         cout << intStack.top() << ' '; // display top element
48         intStack.pop(); // remove top element
49      }
50
51         cout << "\nStack is empty, cannot pop.\n";
52   }
```

```
Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty, cannot pop.

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty, cannot pop.
```

Fig. 15.2 Stack class template `test` program.

Line 31 instantiates `intStack` as a **`Stack<int>`** (pronounced "`Stack` of `int`"). Lines 38–41 repeatedly call `push` (line 39) to place values onto `intStack`. Then, lines 46–49 repeatedly call `isEmpty`, `top` and `pop` to

remove values from `intStack` until it's empty. Again, the output confirms the last-in, first-out order in which the elements are removed.

Though the compiler does not show you the generated code for `Stack<double>` and `Stack<int>`, you can see sample generated code by using the website:

```
https://cppinsights.io
```

This site shows the template instantiations generated by the Clang C++ compiler.[19]

19. The site requires all the code to be pasted into its code pane. To try this with our example, remove the `#include` for `stack.h` in the `main` program and paste class template `Stack`'s code above `main`. Also, remove the `#pragma once` directive.

Defining Class Template Member Functions Outside a Class Template

Member-function definitions can be defined outside a class template definition. In this case, each member-function definition must begin with the same **`template` header** as the class template. Also, you must **qualify each member function with the class name and scope resolution operator**. For example, you'd define the `pop` function outside the class-template definition as

```
template<typename T>
void Stack<T>::pop() {stack.pop_front();}
```

`Stack<T>::` indicates that **`pop` is in class template `Stack<T>`'s scope**. Standard library class templates define some member functions inside and some outside the class-template bodies.

20 15.3 C++20 Function Template Enhancements

In addition to concepts, C++20 added **abbreviated function templates** and **templated lambda expressions**.

20 15.3.1 C++20 Abbreviated Function Templates

Using traditional function-template syntax, we could define a `printContainer` function template with a `template` header as follows:

```
template <typename T>
void printContainer(const T& items) {
   for (const auto& item : items) {
      std::cout << item << " ";
   }
}
```

20 The function template receives a reference to a container (`items`) and uses a range-based `for` statement to display the elements. C++20 **abbreviated function templates** (Fig. 15.3) enable you to **define a function template without the `template` header** (lines 10–14) by using the **`auto` keyword** as the parameter type (line 10).

```
1   // fig15_03.cpp
2   // Abbreviated function template.
3   #include <array>
4   #include <iostream>
5   #include <string>
6   #include <vector>
7
8   // abbreviated function template printContainer displays a
9   // container's elements separated by spaces
10  void printContainer(const auto& items) {
11     for (const auto& item : items) {
12        std::cout << item << " ";
13     }
14  }
15
16  int main() {
17     using namespace std::string_literals; // for string object literals
18
19     std::array ints{1, 2, 3, 4, 5};
20     std::vector strings{"red"s, "green"s, "blue"s};
21
22     std::cout << "ints: ";
23     printContainer(ints);
24     std::cout << "\nstrings: ";
25     printContainer(strings);
26     std::cout << "\n";
27  }
```

```
ints: 1 2 3 4 5
strings: red green blue
```

Fig. 15.3 Abbreviated function template.

Lines 19–20 define an `array` of `int`s and a `vector` of `string`s (line 20), which we initialized with `string` **object literals**. The compiler uses CTAD to **deduce their element types from each initializer list**. In the line 23 and line 25 `printContainer` calls, the compiler infers the parameter type from the containers passed as arguments and generates appropriate template instantiations for each.

Sometimes Traditional Function Template Syntax Is Required

The abbreviated function template syntax is similar to regular function definitions but **is not always appropriate**. Consider the first two lines of Section 5.16's function template `maximum`, which received **three parameters of the same type**:

```
template <typename T>
T maximum(T value1, T value2, T value3) {
```

SE These lines show the correct way to **ensure that all three parameters have the same type**.

If we write `maximum` as an abbreviated function template

```
auto maximum(auto value1, auto value2, auto value3) {
```

the compiler independently infers each `auto` parameter's type based on the corresponding argument. So, `maximum` might receive arguments of three different types.

Using `printContainer` with an Incompatible Type

Err When the compiler attempts to instantiate the `printContainer` function template, errors will occur if

- we pass an **object that is incompatible with a range-based `for` statement** or
- objects of the **container's element type cannot be output with the `<<` operator**.

Such errors are often confusing, as they mention `printContainer` internal implementation details that the caller does not need to know. If the argument is incompatible, it would be better for client-code programmers if the error message(s) simply stated why. In Sections 15.4 and 15.6, we'll show that **C++20 concepts can be used to constrain the types passed to a function template and prevent the compiler from attempting to instantiate the template**. As you'll see, you'll also receive error messages that generally are easier to understand.

20 15.3.2 C++20 Templated Lambdas

In C++20, lambdas can specify template parameters. Consider the following lambda from Fig. 14.16, which we used to calculate the sum of the squares of an `array`'s integers:

```
[](auto total, auto value) {return total + value * value;}
```

SE The compiler independently infers `total`'s and `value`'s (possibly different) types from their arguments. **You can force the lambda to require the same type for both** by using a **templated lambda**:

```
[]<typename T>(T total, T value) {return total + value * value;}
```

Err The template parameter list is placed between the lambda's introducer and the lambda's parameter list. **When you use one template type parameter (`T`) to declare both lambda parameters, the compiler requires both arguments to have the same type**; otherwise, a compilation error occurs.

20 15.4 C++20 Concepts: A First Look

Concepts **Concepts** (briefly introduced in Section 14.2) simplify generic programming. C++ experts say that "Concepts are a revolutionary approach for writing templates"[20] and that "C++20 creates a paradigm shift in the way we use metaprogramming."[21] C++'s creator, Bjarne Stroustrup, says, "Concepts complete C++ templates as originally envisioned" and that they'll "dramatically improve your generic programming and make the current workarounds (e.g., traits classes) and low-level techniques (e.g., `enable_if`-based overloading) feel like error-prone and tedious assembly programming."[22]

20. Bartlomiej Filipek, "C++20 Concepts—A Quick Introduction," May 5 2021. Accessed February 2, 2022. https://www.cppstories.com/2021/concepts-intro/.

21. Inbal Levi, “Exploration of C++20 Meta Programming,” September 29, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=XgrjybKaIV8.
22. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces,” January 31, 2017. Accessed February 2, 2022. http://wg21.link/p0557r0.

As you’ll see, concepts explicitly constrain the arguments specified for a template’s parameters. You’ll use `requires` **clauses** and `requires` **expressions** to specify constraints, which can

- **test attributes of types** (e.g., “Is the type an integer type?”) and
- **test whether types support various operations** (e.g., “Does a type support the comparison operations?”).

The C++ standard provides **74 predefined concepts** (see Section 15.4.3), and you can create custom concepts. Each defines a type’s requirements or a relationship between types.[23] **Concepts can be applied to any parameter of any template and to any use of** `auto`.[24] We’ll discuss concepts in more depth in Section 15.6. Initially, we’ll focus on using concepts to constrain a function template’s type parameters.

23. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces,” January 31, 2017. Accessed February 2, 2022. http://wg21.link/p0557r0.
24. “Placeholder Type Specifiers.” Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/auto.

Motivation and Goals of Concepts

Stroustrup says, “Concepts enable overloading and eliminate the need for a lot of ad-hoc metaprogramming and much metaprogramming scaffolding code, thus significantly simplifying metaprogramming as well as generic programming.”[25]

25. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—3 Using Concepts,” January 31, 2017. Accessed February 2, 2022. http://wg21.link/p0557r0.

Traditionally, **template requirements were implicit**, based on how the template used its arguments in operator expressions, function calls, etc.[26] This was the case in Fig. 15.3’s abbreviated function template `printContainer`. The function definition did not indicate that the argument must be iterable with the range-based `for` statement or that the element type must support the `<<` operator for output. Though such requirements typically would be documented in program comments, **the compiler cannot enforce comments**. To determine that a type was incompatible with a template, the compiler first had to attempt to instantiate the template to “see” that a type did not support the template’s implicit requirements. This led to many, often cryptic compilation errors.

26. Hendrik Niemeyer, “An Introduction to C++20’s Concepts,” July 25, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=N_kPd2OK1L8.

SE **Concepts specify template requirements explicitly in code. They enable the compiler to determine that a type is not compatible with a template before instantiating it**—leading to fewer, more precise error messages, as well as potential compile-time performance improvements.[27]

27. The error messages’ quality and quantity still vary considerably among compilers.

SE **Concepts also enable you to overload function templates with the same signature based on each function template’s requirements.**[28] For example, we’ll define two different overloads of a function template with the same signature:

28. This enables even a class's no-argument constructor to be overloaded with specific constraints.

- one will **support any container with input iterators** and
- the other will be **optimized for containers with random-access iterators**.

15.4.1 Unconstrained Function Template `multiply`

When you call a function, the compiler uses its **overload-resolution rules**[29] and a technique called **argument-dependent lookup (ADL)**[30,31,32,33] to locate all the function definitions that might satisfy the function call. Together, these are known as the **overload set**. This process often includes instantiating function templates based on a function call's argument types. The compiler then chooses the best match from the overload set. **An unconstrained function template does not explicitly specify any requirements.** So the compiler substitutes the call's argument types into the function template's declaration to check whether it is a viable match and, if so, whether it is the best match. Only if it's the best match will the compiler instantiate the template's definition and determine whether the argument types actually support the operations used in the function template's body.

29. "Overload Resolution." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/overload_resolution.
30. C++ Standard, "6.5.4 Argument-Dependent Name Lookup [basic.lookup.argdep]." Accessed February 2, 2022. https://timsong-cpp.github.io/cppwp/n4861/basic.lookup.argdep#:look-up,argument-dependent.
31. "Argument-Dependent Lookup." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/adl.
32. Inbal Levi, "Exploration of C++20 Metaprogramming," September 29, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=XgrjybKaIV8.
33. Arthur O'Dwyer, "What Is ADL?" April 26, 2019. Accessed February 2, 2022. https://quuxplusone.github.io/blog/2019/04/26/what-is-adl/.

Consider the **unconstrained function template** `multiply` (lines 5–6 of Fig. 15.4), which receives **two values of the same type** (`T`) and returns their product.

```
1  // fig15_04.cpp
2  // Simple unconstrained multiply function template.
3  #include <iostream>
4
5  template<typename T>
6  T multiply(T first, T second) {return first * second;}
7
8  int main() {
9     std::cout << "Product of 5 and 3: " << multiply(5, 3)
10       << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0) << "\n";
11 }
```

```
Product of 5 and 3: 15
Product of 7.25 and 2.0: 14.5
```

Fig. 15.4 Simple unconstrained `multiply` function template.

This template has **implicit requirements** that you can infer from the code:

- The parameter types (line 6) are not pointers or references, so the **arguments are received by value**. Similarly, the return type is not a pointer or a reference, so the result is returned by value. Thus, **the arguments' type must support copying** or **moving**.
- The arguments are multiplied, so **the arguments' type must support the binary `*` operator**, either natively (as with built-in types like `int` and `double`) or via operator overloading.

When the compiler instantiates `multiply` for a given type and determines that the arguments are incompatible with the template's implicit requirements, **it eliminates the function from the overload set.** In this example, lines 9 and 10 call `multiply` with two `int` values and two `double` values, respectively. **All numeric types in C++ support this template's implicit requirements**, so the compiler can instantiate `multiply` for each type. Interestingly, **you cannot call `multiply` with arguments of different types,** even if one argument's type can be implicitly converted to the other's. There's one type parameter, so both arguments must have identical types.

Using Incompatible Types with `multiply`

Er What if we pass to `multiply` arguments that **do not support the template's implicit requirements** and there is no other function declaration that better matches the function call? The compiler will generate error messages. For example, the following code attempts to calculate the product of two `string`s:[34]

34. For your convenience, this code is provided in the source-code file as comments at the end of `main`.

```
std::string s1{"hi"};
std::string s2{"bye"};
auto result{multiply(s1, s2)}; // string does not have * operator
```

Class `string` does not support the `*` operator. If we add these lines to Fig. 15.4's `main` and recompile, we get the following error messages from our preferred compilers—we added vertical spacing for readability. **Clang** produces

```
fig15_04.cpp:6:45: error: invalid operands to binary expression
('std::basic_string<char>' and 'std::basic_string<char>')
T multiply(T first, T second) {return first * second;}
                                      ~~~~~ ^ ~~~~~~

fig15_04.cpp:14:16: note: in instantiation of function template
specialization 'multiply<std::basic_string<char>>' requested here
   auto result{multiply(s1, s2)}; // string does not have * operator
               ^
1 error generated.
```

GNU `g++` produces

```
fig15_04.cpp: In instantiation of 'T multiply(T, T) [with T =
std::__cxx11::basic_string<char>]':

fig15_04.cpp:14:24:  required from here

fig15_04.cpp:6:45: error: no match for 'operator*' (operand types are
'std::__cxx11::basic_string<char>' and 'std::__cxx11::basic_string<char>')
```

```
    6 | T multiply(T first, T second) {return first * second;}
      |                                      ~~~~~~^~~~~~~~
```

Visual C++ produces

```
1>fig15_04.cpp
1>c:\pauldeitel\Documents\examples\ch15\fig15_04.cpp(6,45): error C2676: binary '*': 'T'
does not define this operator or a conversion to a type acceptable to the predefined
operator
1>      with
1>      [
1>          T=std::string
1>      ]

1>c:\pauldeitel\Documents\examples\ch15\fig15_04.cpp(14): message : see reference to
function template instantiation 'T multiply<std::string>(T,T)' being compiled
1>      with
1>      [
1>          T=std::string
1>      ]

1>Done building project "concurrencpp_test.vcxproj" -- FAILED.
```

Err These errors are relatively small and straightforward, but this is not typical. More complex templates tend to result in lengthy lists of error messages. For example, passing the wrong kinds of iterators to a standard library algorithm that's not constrained with concepts can yield hundreds of lines of error messages. We tried to compile a program containing only the following two simple statements that attempt to sort a `std::list`:

```
std::list integers{10, 2, 33, 4, 7, 1, 80};
std::sort(integers.begin(), integers.end());
```

A `std::list` has **bidirectional iterators**, but `std::sort` requires **random-access iterators**. One of our preferred compilers generated more than 1,000 lines of error messages for the `std::sort` call! Simply switching to the concept-constrained `std::ranges::sort` algorithm produces far fewer messages and indicates that **random-access iterators** are required.

20 15.4.2 Constrained Function Template with a C++20 Concepts `requires` Clause

Concepts CG **C++20 concepts** enable you to specify constraints. Each is a **compile-time predicate expression** that evaluates to `true` or `false`.[35] The compiler uses constraints to check type requirements **before instantiating templates**. The C++ Core Guidelines recommend

- **specifying concepts for every template parameter**[36] and
- **using the standard's predefined concepts if possible.**[37]

35. C++ Standard, "13.5 Template Constraints." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/temp.constr.

36. C++ Core Guidelines, "T.10: Specify Concepts for All Template Arguments." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-concepts.

37. C++ Core Guidelines, “T.11: Whenever Possible Use Standard Concepts.” Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts.

You can apply concepts to a function template parameter to explicitly state the requirements for the corresponding type arguments. If a type argument satisfies the requirements, the compiler instantiates the template; otherwise, the template is ignored. When compilers do not find a match for a function call, **a benefit of concepts over previous techniques they either partially or wholly replace is that compilers typically produce (possibly far) fewer, clearer and more precise error messages.**

`requires` Clause

20 Figure 15.5 uses a C++20 **`requires` clause** (line 8) to constrain `multiply`'s template parameter `T`. The keyword `requires` is followed by a **constraint expression**, consisting of one or more compile-time `bool` expressions combined with the logical `&&` and `||` operators.

```
1   // fig15_05.cpp
2   // Constrained multiply function template that allows
3   // only integers and floating-point values.
4   #include <concepts>
5   #include <iostream>
6
7   template<typename T>
8      requires std::integral<T> || std::floating_point<T>
9   T multiply(T first, T second) {return first * second;}
10
11  int main() {
12     std::cout << "Product of 5 and 3: " << multiply(5, 3)
13        << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0) << "\n";
14
15     std::string s1{"hi"};
16     std::string s2{"bye"};
17     auto result{multiply(s1, s2)};
18  }
```

```
fig15_05.cpp: In function 'int main()':
fig15_05.cpp:17:24: error: no matching function for call to
'multiply(std::string&, std::string&)'
   17 |  auto result{multiply(s1, s2)};
      |              ~~~~~~~~^~~~~~~~

fig15_05.cpp:9:3: note: candidate:  'template<class T>  requires (integral<T>)
|| (floating_point<T>) T multiply(T, T)'
    9 | T multiply(T first, T second) {return first * second;}
      |   ^~~~~~~~

fig15_05.cpp:9:3: note:  template argument deduction/substitution failed:
fig15_05.cpp:9:3: note: constraints not satisfied
fig15_05.cpp: In substitution of 'template<class T>  requires (integral<T>)
|| (floating_point<T>) T multiply(T, T) [with T = std::__cxx11::basic_string<char>]':
fig15_05.cpp:17:24:  required from here
fig15_05.cpp:9:3:  required by the constraints of 'template<class T>
requires (integral<T>) || (floating_point<T>) T multiply(T, T)'

fig15_05.cpp:8:30: note: no operand of the disjunction is satisfied
    8 |  requires std::integral<T> || std::floating_point<T>
```

```
      |                  ~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~
```

Fig. 15.5 Constrained `multiply` function template that allows only integers and floating-point values. The output shows the `g++` compiler's error messages.

Line 8 specifies that valid `multiply` type arguments must satisfy either of the following concepts, each of which is a **constraint on the type parameter `T`**:

- `std::integral<T>` indicates that `T` **can be any integer data type**.
- `std::floating_point<T>` indicates that `T` **can be any floating-point data type**.

All integral and floating-point types support the arithmetic operators, so we know that these types' values all will work with the `*` operator in line 9. If neither of the preceding constraints is satisfied, the type arguments are incompatible with function template `multiply`, and the compiler will not instantiate the template. If no other functions match the call, the compiler generates error messages.

20 Concepts You'll soon see that **some concepts specify many individual constraints**. The preceding concepts (from header `<concepts>`) are two of **C++20's 74 predefined concepts**.[38]

38. C++ Standard, "Index of Library Concepts." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/conceptindex.

Section 15.4.3 lists the predefined concept categories, the concepts in each and the headers that define them.

Disjunctions and Conjunctions

In line 8, the **logical OR (`||`) operator** forms a **disjunction**. Either or both operands must be `true` for the compiler to instantiate the template. If both are `false`, the compiler ignores the template as a potential match for a call to `multiply`. Function `multiply` defines only one type parameter, so **both operands must have the same type**. Thus, only one concept in line 8 can be `true` for each `multiply` call in this example.

You may form a **conjunction** with the **logical AND (`&&`)** operator to indicate that both operands must be `true` for the compiler to instantiate the template. Disjunctions and conjunctions use **short-circuit evaluation** (Section 4.11.3).

Calling `multiply` with Arguments That Satisfy Its Constraints

Lines 12 and 13 call `multiply` with two `int` values and two `double` values, respectively. When the compiler looks for function definitions that match these calls, it will encounter only the `multiply` function template in lines 7–9. It will check the arguments' types to determine whether they satisfy either concept in line 8's `requires` **clause**. Both arguments in each call satisfy one of the **disjunction**'s requirements, so the compiler will instantiate the template for type `int` in line 12 and type `double` in line 13. Again, **`multiply` has only one type parameter, so both arguments must be the same type**. Otherwise, the compiler will not know which type to use to instantiate the template and will generate errors.

Calling `multiply` with Arguments That Do Not Satisfy Its Constraints

Err Line 17 calls `multiply` with two `string` arguments. When the compiler looks for function definitions that match this call, it will encounter only the `multiply` function template. Next, it will check the arguments' types to determine whether they satisfy either concept listed in line 8's `requires` clause. **The `string` type does not satisfy either**. Also, no other `multiply` functions can receive two `string` arguments, so the compiler generates error messages. Figure 15.5 shows `g++`'s error messages. We highlighted several messages in bold and added vertical spacing for readability. Note the last few lines of this output where the compiler indicated that **`"no operand of the disjunction is`**

`satisfied"` and pointed to the `requires` **clause** in line 8. For this example, Visual C++ produced the simplest error messages of our three preferred compilers:

```
1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(17,16): error
C2672: 'multiply': no matching overloaded function found

1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(17,31): error
C7602: 'multiply': the associated constraints are not satisfied

1>c:\pauldeitel\Documents\examples\ch15\fig15_05.cpp(9): message :
see declaration of 'multiply'
```

20 15.4.3 C++20 Predefined Concepts

Concepts The C++ standard's "Index of library concepts"[39] alphabetically lists the concepts defined in the standard. The following table lists all the standard concepts by header—`<concepts>`, `<iterator>`, `<ranges>`, `<compare>` and `<random>`. We divided the `<concepts>` header's 31 concepts into the subcategories **core language concepts**, **comparison concepts**, **object concepts** and **callable concepts**. Many concepts have self-explanatory names. For details on each, see the corresponding header's page at cppreference.com.

39. C++ Standard, "Index of Library Concepts." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/conceptindex.

74 predefined C++20 concepts

<concepts>header core language concepts

`assignable_from`	`default_initializable`	`same_as`
`common_reference_with`	`derived_from`	`signed_integral`
`common_with`	`destructible`	`swappable`
`constructible_from`	`floating_point`	`swappable_with`
`convertible_to`	`integral`	`unsigned_integral`
`copy_constructible`	`move_constructible`	

<concepts> header comparison concepts

`equality_comparable`	`totally_ordered`	`totally_ordered_with`
`equality_comparable_with`		

74 predefined C++20 concepts

<concepts> header object concepts

copyable	regular	semiregular
movable		

<concepts> header callable concepts

equivalence_relation	predicate	relation
invocable	regular_invocable	strict_weak_order

<iterator> header concepts

bidirectional_iterator	indirectly_copyable_storable	input_or_output_iterator
contiguous_iterator	indirectly_movable	mergeable
forward_iterator	indirectly_movable_storable	output_iterator
incrementable	indirectly_readable	permutable
indirect_binary_predicate	indirectly_regular_unary_	random_access_iterator
indirect_equivalence_relation	invocable	sentinel_for
indirect_strict_weak_order	indirectly_swappable	sized_sentinel_for
indirect_unary_predicate	indirectly_unary_invocable	sortable
indirectly_comparable	indirectly_writable	weakly_incrementable
indirectly_copyable	input_iterator	

<ranges>header concepts

bidirectional_range	forward_range	random_access_range
borrowed_range	input_range	sized_range
common_range	output_range	view_range
contiguous_range	range	viewable_range

74 predefined C++20 concepts

<compare> header concepts

`three_way_comparable` `three_way_comparable_with`

<random> header concept

`uniform_random_bit_generator`

15.5 Type Traits

11 C++11 introduced the **`<type_traits>` header**[40] for

40. “Standard Library Header `<type_traits>`.” Accessed February 2, 2022. https://en.cppreference.com/w/cpp/header/type_traits.

- testing at compile-time whether types have various traits and
- generating template code based on those traits.

For example, you could check whether a type is

- a **fundamental type like** `int` (using the type trai t `std::is_fundamental`) vs.
- a **class type** (using the type trait `std::is_class`)

20 and use different template code to handle each case. Each subsequent C++ version has added more type traits, and a few have been deprecated or removed. Most recently, **C++20 added 10 type traits**.

Using Type Traits Before C++20

Before concepts, you’d use type traits in unconstrained template definitions to check whether type arguments satisfied a template’s requirements. As with concepts, these checks were performed at compile-time **but during template instantiation**, often leading to many cryptic error messages. On the other hand, **the compiler tests concepts before instantiating templates**, typically resulting in fewer, more precise error messages than when you use type traits in unconstrained templates.

20 C++20 Predefined Concepts Often Use Type Traits

C++20 concepts are often implemented in terms of type traits. For example:

- the **concept** `std::integral` is implemented in terms of the **type trait** `std::is_integral`,
- the **concept** `std::floating_point` is implemented in terms of the **type trait** `std::is_floating_point` and
- the **concept** `std::destructible` is implemented in terms of the **type trait** `std::is_nothrow_destructible`.

Demonstrating Type Traits

Figure 15.6 shows **type traits** that correspond to the concepts in Fig. 15.5.

```cpp
// fig15_06.cpp
// Using type traits to test whether types are
// integral types, floating-point types or arithmetic types.
#include <fmt/format.h>
#include <iostream>
#include <string>
#include <type_traits>

int main() {
  std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n{}{}\n\n",
     "CHECK WITH TYPE TRAITS WHETHER TYPES ARE INTEGRAL",
     "std::is_integral<int>::value: ", std::is_integral<int>::value,
     "std::is_integral_v<int>: ", std::is_integral_v<int>,
     "std::is_integral_v<long>: ", std::is_integral_v<long>,
     "std::is_integral_v<float>: ", std::is_integral_v<float>,
     "std::is_integral_v<std::string>: ",
     std::is_integral_v<std::string>);

  std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n{}{}\n\n",
     "CHECK WITH TYPE TRAITS WHETHER TYPES ARE FLOATING POINT",
     "std::is_floating_point<float>::value: ",
     std::is_floating_point<float>::value,
     "std::is_floating_point_v<float>: ",
     std::is_floating_point_v<float>,
     "std::is_floating_point_v<double>: ",
     std::is_floating_point_v<double>,
     "std::is_floating_point_v<int>: ",
     std::is_floating_point_v<int>,
     "std::is_floating_point_v<std::string>: ",
     std::is_floating_point_v<std::string>);

  std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n{}{}\n",
     "CHECK WITH TYPE TRAITS WHETHER TYPES CAN BE USED IN ARITHMETIC",
     "std::is_arithmetic<int>::value: ", std::is_arithmetic<int>::value,
     "std::is_arithmetic_v<int>: ", std::is_arithmetic_v<int>,
     "std::is_arithmetic_v<double>: ", std::is_arithmetic_v<double>,
     "std::is_arithmetic_v<std::string>: ",
     std::is_arithmetic_v<std::string>);
}
```

```
CHECK WITH TYPE TRAITS WHETHER TYPES ARE INTEGRAL
std::is_integral<int>::value: true
std::is_integral_v<int>: true
std::is_integral_v<long>: true
std::is_integral_v<float>: false
std::is_integral_v<std::string>: false

CHECK WITH TYPE TRAITS WHETHER TYPES ARE FLOATING POINT
std::is_floating_point<float>::value: true
std::is_floating_point_v<float>: true
std::is_floating_point_v<double>: true
std::is_floating_point_v<int>: false
std::is_floating_point_v<std::string>: false
```

```
CHECK WITH TYPE TRAITS WHETHER TYPES CAN BE USED IN ARITHMETIC
std::is_arithmetic<int>::value: true
std::is_arithmetic_v<int>: true
std::is_arithmetic_v<double>: true
std::is_arithmetic_v<std::string>: false
```

Fig. 15.6 Using type traits to test whether types are integral types, floating-point types or arithmetic types.

Each type-trait class in this example has a `static constexpr bool` member named `value`. An expression like line 12's

```
std::is_integral<int>::value
```

evaluates to `true` at compile-time if the type in angle brackets (`int`) is an integral type; otherwise, it evaluates to `false`.

17 14 The notation **`::value`** is commonly used with type-trait classes to access their `true` or `false` values, so **C++17 added convenient shorthands for using type trait values.** These are defined as **variable templates** (a C++14 feature) with **names ending in `_v`.**[41] Just as function templates specify groups of related functions and class templates specify groups of related classes, variable templates specify groups of related variables. **You instantiate a variable template to use it.** The variable template corresponding to

41. C++ Core Guidelines, "T.142: Use Template Variables to Simplify Notation." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-var.

```
std::is_integral<int>::value
```

is defined in the C++ standard as[42]

42. C++ Standard, "20.15.3 Header `<type_traits>` Synopsis." Accessed February 2, 2022. https://tim-song-cpp.github.io/cppwp/n4861/meta#type.synop.

```
template<class T>
inline constexpr bool is_integral_v = is_integral<T>::value;
```

So, the expression **`std::is_integral_v<int>`** is equivalent to

```
std::is_integral<int>::value
```

Lines 12, 22 and 34 test type traits and display their `value` members to show the results. The program's other tests use the more convenient **`_v` variable templates**:

- Lines 13, 14, 15 and 17 use **`std::is_integral_v`** to check whether various types are **integer types**.
- Lines 24, 26, 28 and 30 use **`std::is_floating_point_v`** to check whether various types are **floating-point types**.
- Lines 35, 36 and 38 use **`std::is_arithmetic_v`** to check whether various types are **arithmetic types** that could be used in arithmetic expressions.

The following table lists the `<type_traits>` header's type traits and supporting functions by category. The new features added in C++14 (two new items), 17 (14 new items) and 20 (10 new items) are indicated with superscript version numbers. For details on each, see

https://en.cppreference.com/w/cpp/header/type_traits

This table lists the type traits in the same categories and order as cppreference.com.

Type traits by category

Helper classes

bool_constant[17]
integral_constant
false_type
true_type

Primary type categories

is_void
is_null_pointer[14]
is_integral
is_floating_point
is_array
is_enum
is_union
is_class
is_function
is_pointer
is_lvalue_reference
is_rvalue_reference
is_member_object_pointer
is_member_function_pointer

Composite type categories

is_fundamental
is_arithmetic
is_scalar
is_object
is_compound
is_reference
is_member_pointer

Type properties

has_unique_object_representations[17]
is_const
is_volatile
is_trivial
is_trivially_copyable
is_standard_layout
is_empty
is_polymorphic
is_abstract
is_final[14]
is_aggregate[17]
is_signed
is_unsigned
is_bounded_array[20]
is_unbounded_array[20]
is_scoped_enum[23]

Supported operations

is_constructible
is_trivially_constructible
is_nothrow_constructible
is_default_constructible
is_trivially_default_constructible
is_nothrow_default_constructible
is_copy_constructible
is_trivially_copy_constructible
is_nothrow_copy_constructible
is_move_constructible
is_trivially_move_constructible
is_nothrow_move_constructible
is_assignable
is_trivially_assignable
is_nothrow_assignable
is_copy_assignable
is_trivially_copy_assignable
is_nothrow_copy_assignable
is_move_assignable
is_trivially_move_assignable
is_nothrow_move_assignable
is_destructible
is_trivially_destructible
is_nothrow_destructible
has_virtual_destructor
is_swappable_with[17]
is_swappable[17]
is_nothrow_swappable_with[17]
is_nothrow_swappable[17]

Property queries

Type traits by category

alignment_of	rank	extent

Type relationships

is_same	is_layout_compatible[20]	is_invocable_r[17]
is_base_of	is_pointer_interconvertible_base_of[20]	is_nothrow_invocable[17]
is_convertible	is_invocable[17]	is_nothrow_invocable_r[17]
is_nothrow_convertible[20]		

Const-volatility specifiers

remove_cv	remove_volatile	add_const
remove_const	add_cv	add_volatile

References

remove_reference	add_lvalue_reference	add_rvalue_reference

Pointers

remove_pointer	add_pointer

Sign modifiers

make_signed	make_unsigned

Arrays

remove_extent	remove_all_extents

Miscellaneous transformations

aligned_storage	conditional	underlying_type
aligned_union	common_type	invoke_result[17]
decay	common_reference[20]	void_t[17]
remove_cvref[20]	basic_common_reference[20]	type_identity[20]
enable_if		

Operations on traits

conjunction[17]	disjunction[17]	negation[17]

Functions Member relationships

is_pointer_interconvertible_with_class[20]	is_corresponding_member[20]

20 15.6 C++20 Concepts: A Deeper Look

Now that we've introduced how to constrain a template parameter via the `requires` **clause** and **predefined concepts**, we'll create a **custom concept** that aggregates two predefined ones and present additional concepts features.

Concepts *15.6.1 Creating a Custom Concept*

C++20's **predefined concepts often aggregate multiple constraints, sometimes including those from other predefined concepts**, effectively creating what some developers refer to as a **type category**.[43] You also can use templates to define your own custom concepts. Let's create a **custom concept** that aggregates the predefined concepts `std::integral` and `std::floating_point` we used in Fig. 15.5 to constrain the `multiply` function template:

43. Inbal Levi, "Exploration of C++20 Metaprogramming," September 29, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=XgrjybKaIV8.

```
template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
```

20 A concept begins with a `template` header followed by

- the C++20 keyword `concept`,
- the **concept name** (`Numeric`),
- an **equal sign** (`=`) and
- the **constraint expression** (`std::integral<T> || std::floating_point<T>`).

The **constraint expression** is a compile-time logical expression that determines whether a template argument satisfies a particular set of requirements—in this case, whether it's an integer or floating-point type. Constraint expressions commonly include **predefined concepts**, **type traits** and, as you'll see in Section 15.6.5, `requires` **expressions** that enable you to specify other requirements.

The `Numeric` concept's `template` header has one type parameter, representing the type to test.

Concepts with multiple template parameters also can specify relationships between types.[44] For example, if a template has two type parameters, you can **use the predefined concept** `std::same_as` **to ensure two type arguments have the same type**, as in:

44. Bjarne Stroustrup, "Concepts: The Future of Generic Programming—3.1 Specifying Template Interfaces," January 31, 2017. Accessed February 2, 2022. http://wg21.link/p0557r0.

```
template<typename T, typename U>
   requires std::same_as<T, U>
// rest of template definition
```

In Section 15.12, we'll introduce **variadic templates** that can have any number of (type or non-type) template arguments. There, we'll use `std::same_as` in a variadic function template that requires all its arguments to have the same type.

Once you define a concept, you can use it to **constrain a template parameter** in one of four ways. We show the first three here and the fourth in Section 15.6.3.

requires Clause Following the template Header

Any concept can be placed in a `requires` clause **following the `template` header**, as we did in Fig. 15.5. Here's our `multiply` function template using the **custom concept `Numeric<T>`**:

```
template<typename T>
   requires Numeric<T>
T multiply(T first, T second) {return first * second;}
```

`requires` Clause Following a Function Template's Signature

You also can place the **`requires` clause after a function template's signature and before the function template's body**, as in:

```
template<typename T>
T multiply(T first, T second) requires Numeric<T> {
   return first * second;
}
```

A **trailing `requires` clause** must be used in two scenarios:[45]

45. "What Is the Difference Between the Three Ways of Applying Constraints to a Template?" Accessed February 2, 2022. https://stackoverflow.com/a/61875483. [Note: The original stackoverflow.com question mentioned three ways of applying constraints to a template, but there are four and each is mentioned in the cited answer.]

- A member function defined in a class template's body does not have a `template` header, so you must use a trailing `requires` clause.
- To use a function template's parameter names in a constraint, you must use a trailing `requires` clause, so the parameter names are in scope before the compiler evaluates the `requires` clause.

Concept as a Type in the `template` Header

CG When you have a single concept-constrained type parameter, the C++ Core Guidelines recommend using the concept name in place of `typename` in the `template` header.[46] Doing so simplifies the template definition by **eliminating the `requires` clause**:

46. C++ Core Guidelines, "T.13: Prefer the Shorthand Notation for Simple, Single-Type Argument Concepts." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-shorthand.

```
template<Numeric T>
Number multiply(T first, T second) {return first * second;}
```

Err Now, each parameter must satisfy the **concept `Numeric`'s** requirements. The function template still has only one type parameter, so the function's arguments must have the same type; otherwise, a compilation error occurs.

Concepts *15.6.3 Using Concepts in Abbreviated Function Templates*

SE Figure 15.3 introduced **abbreviated function templates** with the parameters declared `auto` so the compiler can infer the function's parameter types. **Anywhere you can use `auto` in your code, you can precede it with a concept name to constrain the allowed types.**[47]

47. "Placeholder Type Specifiers." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/auto.

This includes

- abbreviated function template parameter lists,
- `auto` specified as a function's return type,
- `auto` local-variable definitions that infer a variable's type from an initializer, and
- generic lambda expressions.

Figure 15.7 reimplements `multiply` as an **abbreviated function template**. In this case, we used **constrained `auto`** for each parameter, using our **`Numeric` concept** to restrict the types we can pass as arguments. We also **used `auto` as the return type**, so the compiler will infer it from the type of the expression in line 12.

```
1  // fig15_07.cpp
2  // Constrained multiply abbreviated function template.
3  #include <concepts>
4  #include <iostream>
5
6  // Numeric concept aggregates std::integral and std::floating_point
7  template<typename T>
8  concept Numeric = std::integral<T> || std::floating_point<T>;
9
10 // abbreviated function template with constrained auto
11 auto multiply(Numeric auto first, Numeric auto second) {
12    return first * second;
13 }
14
15 int main() {
16   std::cout << "Product of 5 and 3: " << multiply(5, 3)
17      << "\nProduct of 7.25 and 2.0: " << multiply(7.25, 2.0)
18      << "\nProduct of 5 and 7.25: " << multiply(5, 7.25) << "\n";
19 }
```

```
Product of 5 and 3: 15
Product of 7.25 and 2.0: 14.5
Product of 5 and 7.25: 36.25
```

Fig. 15.7 Constrained `multiply` abbreviated function template.

Err The key difference between this **abbreviated function template** and the constrained `multiply` function templates in Section 15.4.2 is that the compiler treats each `auto` in line

11 as a **separate template type parameter**. Thus, `first` and `second` **can have different data types**, as in line 18, which passes an `int` and a `double`. Our previous **concept-constrained version** of `multiply` would generate compilation errors for arguments of different types. Lines 11–13 are actually equivalent to a template with **two type parameters**:

```
template<Numeric T1, Numeric T2>
auto multiply(T1 first, T2 second) {return first * second;}
```

SE If you require the same type for two or more parameters,

- use a regular function template rather than an abbreviated function template, and
- use the same type parameter name for every function parameter that must have the same type.

Concepts *15.6.4 Concept-Based Overloading*

SE Function templates and overloading are intimately related. **When overloaded functions perform syntactically identical operations on different types, they can be expressed more compactly and conveniently using function templates**. You can then write function calls with different argument types and let the compiler instantiate the template appropriately for each function call. The instantiations all have the same function name, so the compiler uses **overload resolution** to invoke the proper one.

Matching Process for Overloaded Functions

When determining which function to call, the compiler looks at functions and function templates to locate an existing function or generate a function-template specialization that matches the call:

- If there are no matches, the compiler issues an error message.
- If there are **multiple matches** for the function call, the compiler attempts to determine the **best match**.
- Err If there's **more than one best match**, the call is **ambiguous**, and the compiler issues an error message.[48]

48. The compiler's process for resolving function calls is complex. The complete details are discussed in the C++ Standard, "12.2 Overload Resolution." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/over.match.

Overloading Function Templates

You also may **overload function templates**. For example, you can provide other function templates with different signatures. A function template also can be overloaded by providing non-template functions with the same function name but different parameters.

Overloading Function Templates Using Concepts

20 **With C++20, you can use concepts in function templates to select overloads based on type requirements.**[49] Consider the standard functions `std::distance` and `std::advance` from the `<iterator>` header—each operates on a container via iterators:

49. Bjarne Stroustrup, “Concepts: The Future of Generic Programming—6 Concept Overloading,” January 31, 2017. Accessed February 2, 2022. http://wg21.link/p0557r0.

- `std::distance` calculates the number of elements between two iterators.[50]

50. “`std::distance`.” Accessed February 2, 2022. https://en.cppreference.com/w/cpp/iterator/distance.

- `std::advance` advances an iterator *n* positions from its current location.[51]

51. “`std::advance`.” Accessed February 2, 2022. https://en.cppreference.com/w/cpp/iterator/advance.

Each of these can be implemented as **O(n)** operations:

- `std::distance` can use `++` to iterate from a begin iterator up to but not including an end iterator and count the number of increments (*n*).
- `std::advance` can loop *n* times, incrementing the iterator once per loop iteration.

Some algorithms can be optimized for specific iterator types. Both `std::distance` and `std::advance` can be implemented as **O(1)** operations for **random-access iterators**:

- `std::distance` can use `operator-` to subtract a begin iterator from an end iterator to **calculate the number of items between the iterators in one operation**.
- `std::advance` can use `operator+` to add an integer *n* to an iterator, **advancing the iterator n positions in one operation**.

Figure 15.8 implements **overloaded `customDistance` function templates**. Each requires two iterator arguments and calculates the number of elements between them. We use **concept-based overloading** (also called **concept overloading**) to enable the compiler to **choose between the overloads based on the concepts we use to constrain each template’s parameters.**

```
1   // fig15_08.cpp
2   // Using concepts to select overloads.
3   #include <array>
4   #include <iostream>
5   #include <iterator>
6   #include <list>
7
8   // calculate the distance (number of items) between two iterators
9   // using input iterators; requires incrementing between iterators,
10  // so this is an O(n) operation
11  template <std::input_iterator Iterator>
12  auto customDistance(Iterator begin, Iterator end) {
13      std::cout << "Called customDistance with input iterators\n";
14      std::ptrdiff_t count{0};
15
16    // increment from begin to end and count number of iterations
17  for (auto& iter{begin}; iter != end; ++iter) {
18    ++count;
19  }
20
21  return count;
22 }
23
24  // calculate the distance (number of items) between two iterators
25  // using random-access iterators and an O(1) operation
26  template <std::random_access_iterator Iterator>
27  auto customDistance(Iterator begin, Iterator end) {
28     std::cout << "Called customDistance with random-access iterators\n";
29     return end - begin; // returns a std::ptrdiff_t value
```

```
30 }
31
32   int main() {
33      std::array ints1{1, 2, 3, 4, 5}; // has random-access iterators
34      std::list ints2{1, 2, 3}; // has bidirectional iterators
35
36      auto result1{customDistance(ints1.begin(), ints1.end())};
37      std::cout << "ints1 number of elements: " << result1 << "\n";
38      auto result2{customDistance(ints2.begin(), ints2.end())};
39      std::cout << "ints2 number of elements: " << result2 << "\n";
40 }
```

```
Called customDistance with random-access iterators
ints1 number of elements: 5
Called customDistance with input iterators
ints2 number of elements: 3
```

Fig. 15.8 Using concepts to select overloads.

To distinguish between the function templates, we use the predefined `<iterator>` header concepts `std::random_access_iterator` and `std::input_iterator`:

- The **O(n) `customDistance` function** in lines 11–22 requires two arguments that satisfy the `std::input_iterator` concept.
- The **O(1) `customDistance` function** in lines 26–30 requires two arguments that satisfy the `std::random_access_iterator` concept.

C++ uses the type `std::ptrdiff_t` (line 14) to represent the difference between two pointers or two iterators. The `std::distance` algorithm returns this type, so we do as well in our `customDistance` implementations.

SE Lines 33–34 define an `array` and a `list`. Line 36 calls `customDistance` for the `array ints1`, which has **random-access iterators**. Our function that requires input iterators could receive random-access iterators. However, **when multiple function templates satisfy a function call, the compiler calls the most constrained version.**[52,53] So line 36 calls the **O(1) version of `customDistance`** (lines 26–30). Based on their definitions, the compiler knows `random_access_iterator` is more constrained than `input_iterator`. For the call in line 38, a `list`'s **bidirectional iterators** do not satisfy the constraints of the function template in lines 26–30. So, that version is eliminated from consideration, and the slower $O(n)$ `customDistance` in lines 11–22 is called.

52. Hendrik Niemeyer, "An Introduction to C++20's Concepts," July 25, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=N_kPd2OK1L8.
53. C++ Standard, "13.5 Template Constraints." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/temp.constr.

15.6.5 requires Expressions *Concepts*

CG The **C++ Core Guidelines recommend using standard concepts.**[54] For custom requirements that cannot be expressed via standard concepts in a **`requires` clause**, you can use a **`requires` expression**, which has two forms:

54. C++ Core Guidelines, "T.11: Whenever Possible Use Standard Concepts." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-std-concepts.

```
requires {
   requirement-definitions
```

```
}
```
or

```
requires (parameter-list) {
   requirements-definitions that optionally use the parameters
}
```
The *parameter-list* looks like a function's parameter list but **cannot have default arguments**. The compiler uses a `requires` expression's parameters only to check whether types satisfy the requirements defined in the expression's braces. The braces may contain any combination of the four requirement types described below—**simple**, **type**, **compound** and **nested**. Each requirement ends with a semicolon (`;`).

Simple Requirements

20 A **simple requirement** checks whether an expression is valid. Consider the definition of the `<ranges>` header's **`range` concept**, which checks whether an object's type represents a **C++20 range** with a **begin iterator** and an **end sentinel**:[55]

55. C++ Standard, "24.4.2 Ranges." Accessed February 2, 2022. https://timsong-cpp.github.io/cppwp/n4861/ranges#range.range.

```
1   template<class T>
2   concept range =
3       requires(T& t) {
4           std::ranges::begin(t);
5           std::ranges::end(t);
6       };
```
The requirements can reference the expression's parameter(s) and template parameter(s) from the `template` header. This **`range` concept** defines a `T&` parameter `t`. Lines 4–5 define **simple requirements** indicating that `t` is a `range` only if we can get `t`'s **begin iterator** and **end sentinel** by passing `t` to functions **`std::ranges::begin`** and **`std::ranges::end`**, respectively. If either of these **simple requirements** does not compile, then `t` is not a `range`.

Simple requirements can specify operator expressions. The following concept specifies operations that would be expected of any integer or floating-point arithmetic type:

```
template<class T>
concept ArithmeticType =
   requires(T a, T b) {
     a + b;
     a - b;
     a * b;
     a / b;
     a += b;
     a -= b;
     a *= b;
     a /= b;
  };
```
We did not include modulus (`%`) because it requires integer operands. Of course, built-in arithmetic types support more operations, such as implicit conversions between types, so a "real" **`ArithmeticType` concept** would be more elaborate. The **type trait `is_arithmetic`** tests for built-in arithmetic types—for custom-class types, it always evaluates to `false`.

Type Requirements

A **type requirement** starts with `typename` followed by a type and determines whether the specified type is valid. For example, if your code requires that a type argument have a nested type called `value_type` (which is the case for standard library containers), you might define a concept with a type requirement like

```
template<typename T>
concept HasValueType = requires {
   typename T::value_type;
 };
```

A type `T` would satisfy `HasValueType` only if `T::value_type` is a valid type. If type `T` does not contain a nested `value_type`, this requirement would evaluate to `false`.

Compound Requirements

A **compound requirement** allows you to specify an expression that also has requirements on its result. Such requirements have the form

```
{ expression } -> return-type-requirement
```

For example, the standard library `<iterator>` header's **`incrementable` concept** contains the following `requires` expression, specifying that an `incrementable` iterator must support the postincrement operator (`++`):[56]

56. C++ Standard, "23.3.4.5 Concept `incrementable`." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/iterator.concepts#iterator.concept.inc.

```
requires(I i) {
  { i++ } -> same_as<I>;
 }
```

The `->` notation indicates a requirement on the `i++` expression's result. In this case, the result must have the same type (`I`) as the object `i`'s type. The right brace in a compound requirement optionally may be followed by `noexcept` to indicate that the expression must be `noexcept`. When we introduced **`std::same_as`** earlier, we used two template arguments, but we used only one here. When you specify a constraint on an expression's result, the compiler inserts the expression's type into the constraint as the first type parameter. So there actually are two type parameters in the preceding constraint—in this case, both are type `I`.[57]

57. C++ Standard, "13.2 Template Parameters." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/temp.param#4.

An **`incrementable`** object also must support the **concept `weakly_incrementable`**, which among its other requirements contains the following **compound requirement**:

```
{ ++i } -> same_as<I&>;
```

So, **`incrementable` objects also must support preincrementing, and the expression's result must have the same type as a reference to the type parameter `I`.**

Nested Requirement

A nested requirement is simply a `requires` clause nested in a `requires` expression. You'd use nested requirements to apply existing concepts and type traits to the `requires` expression's parameters.

`requires requires`—Ad-Hoc Constraints

A `requires` expression placed directly in a `requires` clause is an ad-hoc constraint. In the following `printRange` function template, we copied the `std::ranges::range` concept's `requires` expression and placed it directly in a `requires` clause:

```
template<typename T>

   requires requires(const T& t) {

       std::ranges::begin(t);
       std::ranges::end(t);
  }
void printRange(const T& range) {
  for (const auto& item : range) {
     std::cout << item << " ";
  }
}
```

The notation `requires requires` is correct:

- The first `requires` introduces the `requires` clause.
- The second `requires` introduces the `requires` expression.

SE The benefit of an ad-hoc constraint is that if you need it only once, you can define it where it's used. **Named concepts are preferred**.

15.6.6 C++20 Exposition-Only Concepts

20 Concepts Throughout the C++ standard document, the phrases "**for the sake of exposition**" or "**exposition only**" appear over 400 times. These are displayed in *italics* and indicate **items used only for discussion purposes.**[58] They often show how something can be implemented. For example, the standard uses the following **exposition-only has-arrow concept** in the ranges library to describe an iterator type that supports the operator `->`:[59]

58. "Exposition-Only in the C++ Standard?" Answered December 28, 2015. Accessed February 2, 2022. https://stackoverflow.com/questions/34493104/exposition-only-in-the-c-standard.

59. C++ Standard, "24.5.1 Helper Concepts." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/range.utility.

```
template<class I>

  concept has-arrow =
    input_iterator<I> && (is_pointer_v<I> ||
      requires(I i) { i.operator->(); });
```

Similarly, the standard uses the following **exposition-only decrementable concept** in the ranges library to describe iterator types that support the `--` operator:[60]

60. C++ Standard, “24.6.4.2 Class Template `iota_view`.” Accessed February 2, 2022. https://tim-song-cpp.github.io/cppwp/n4861/range.factories#range.iota.view-2.

```
template<class I>
  concept decrementable =
     incrementable<I> && requires(I i) {
         { --i } -> same_as<I&>;
         { i-- } -> same_as<I>;
   };
```

You may wonder why *decrementable* requires *incrementable*. Recall that all iterators support `++`, so a *decrementable* iterator also must be *incrementable*.

The standard uses **31 exposition-only concepts**, which are shown in the following table. You can find each through the standard’s “Index of library concepts.”[61]

61. C++ Standard, “Index of Library Concepts.” Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/conceptindex.

31 exposition-only concepts

C++20 concepts library

boolean-testable	*same-as-impl*
boolean-testable-impl	*weakly-equality-comparable-with*

Iterators library

can-reference	*cpp17-input-iterator*	*dereferenceable*
cpp17-bidirectional-iterator	*cpp17-iterator*	*indirectly-readable-impl*
cpp17-forward-iterator	*cpp17-random-access-iterator*	*simple-view*

C++20 ranges library

advanceable	*has-tuple-element*	*pair-like-convertible-from*
convertible-to-non-slicing	*iterator-sentinel-pair*	*stream-extractable*
decrementable	*not-same-as*	*tiny-range*
has-arrow	*pair-like*	

Comparisons in the language support library

compares-as	*partially-ordered-with*

Memory library

no-throw-forward-iterator	*no-throw-input-iterator*	*no-throw-sentinel*
no-throw-forward-range	*no-throw-input-range*	

15.6.7 Techniques Before C++20 Concepts: SFINAE and Tag Dispatch

With each new C++ standard, metaprogramming gets more powerful and convenient. There's a history of several technologies that led to C++20 concepts, including **SFINAE**, **tag dispatch** and `constexpr if` (Section 15.13.3 shows `constexpr`). For a nice overview of the progression through these technologies, see the blog post, "Notes on C++ SFINAE, Modern C++ and C++20 Concepts."[62]

62. Bartlomiej Filipek, "Notes on C++ SFINAE, Modern C++ and C++20 Concepts," April 20, 2020. Accessed February 2, 2022. https://www.bfilipek.com/2016/02/notes-on-c-sfinae.html.

SFINAE—Substitution Failure Is Not an Error

Earlier, we mentioned that when you call a function, the compiler locates all the functions that might satisfy the function call—known as the **overload set**. From these overloads, the compiler chooses the best match. This process often includes **instantiating function templates** based on a function call's argument types.

Before concepts, developers had to be significantly more familiar with template specialization rules. Since template debugging is nontrivial, this made the code more error-prone. Template metaprogramming techniques involving `std::enable_if` were commonly used with **type traits** to check if a function-template argument satisfied a template's requirements. If not, the compiler would generate invalid code. It would then ignore that code, removing it from the overload set. **SFINAE (substitution failure is not an error)**[63,64,65] describes how the compiler discards invalid template code as it determines the correct function to call. SFINAE prevents the compiler from immediately generating potentially lengthy lists of errors when first instantiating a template. The compiler generates error messages only if it cannot find a match for the function call in the overload set.

63. "SubstitutionFailure Is Not an Error." Wikipedia. Wikimedia Foundation. Accessed February 2, 2022. https://en.wikipedia.org/wiki/Substitution_failure_is_not_an_error.

64. Filipek, "Notes on C++ SFINAE, Modern C++ and C++20 Concepts."

65. David Vandevoorde and Nicolai M. Josuttis, *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

Tag Dispatch

20 You can tell the compiler the version of an overloaded function to call based not only on template type parameters but also on properties of those types using the **tag-dispatch**[66] technique. Bjarne Stroustrup—in his paper "Concepts: The Future of Generic Programming"—refers to properties of types as "concepts" and discusses how C++20 **concept-based overloading** (Section 15.6.4) can replace tag dispatch.[67]

66. "Generic Programming—Tag Dispatching." Accessed February 2, 2022. https://www.boost.org/community/generic_programming.html#tag_dispatching.

67. Bjarne Stroustrup, "Concepts: The Future of Generic Programming (Section 6)." Accessed February 2, 2022. https://www.stroustrup.com/good_concepts.pdf.

20 Concepts 11 Concepts produce compile-time `bool` values, which you can test at compile-time with a C++11 **`static_assert` declaration**.[68] **`static_assert`** was developed to add compile-time assertion support for **reporting incorrect usage of template libraries.**[69] Figure 15.9 tests our custom `Numeric` concept from Section 15.6.1 in a `multiply` function template that is not concept constrained.[70]

68. C++ Core Guidelines, "T.150: Check That a Class Matches a Concept Using `static_assert`." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-check-class.

69. Robert Klarer, John Maddock, Beman Dawes and Howard Hinnant, "Proposal to Add Static Assertions to the Core Language (Revision 3)," October 20, 2004. Accessed February 2, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html.

70. With concepts, you do not need to use `static_assert` as shown here.

```
// fig15_09.cpp
// Testing custom concepts with static_assert.
#include <iostream>
#include <string>

template<typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;

template<typename T>
auto multiply(T a, T b) {
  static_assert(Numeric<T>);
  return a * b;
}

int main() {
   using namespace std::string_literals;
   multiply(2, 5); // OK: int is Numeric
   multiply(2.5, 5.5); // OK: double is Numeric
   multiply("2"s, "5"s); // error: string is not Numeric
}
```

```
fig15_09.cpp:11:4: error: static_assert failed
   static_assert(Numeric<T>);
   ^             ~~~~~~~~~~

fig15_09.cpp:19:4: note: in instantiation of function template specialization
'multiply<std::basic_string<char>>' requested here
   multiply("2"s, "5"s); // error: string is not Numeric
    ^

fig15_09.cpp:11:18: note: because 'std::basic_string<char>' does not satisfy
'Numeric'
   static_assert(Numeric<T>);
                 ^

fig15_09.cpp:7:24: note: because 'std::basic_string<char>' does not satisfy
'integral'
concept Numeric = std::integral<T> || std::floating_point<T>;
```

```
                      ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:102:24: note: because 'is_integral_v<std::basic_string<char> >'
evaluated to false
    concept integral = is_integral_v<_Tp>;
                       ^

fig15_09.cpp:7:44: note: and 'std::basic_string<char>' does not satisfy
'floating_point'
concept Numeric = std::integral<T> || std::floating_point<T>;
                                           ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:111:30: note: because 'is_floating_point_v<std::basic_string<char> >'
evaluated to false
    concept floating_point = is_floating_point_v<_Tp>;
                             ^

fig15_09.cpp:12:13: error: invalid operands to binary expression
('std::basic_string<char>' and 'std::basic_string<char>')
   return a * b;
          ~ ^ ~

2 errors generated.
```

Fig. 15.9 Testing custom concepts with `static_assert`.

Err When the **`static_assert`** argument is `false`, the compiler outputs an error message that tells you what and where the error is. If the **`static_assert`** argument is `true`, the compiler does not output any messages—it simply continues compiling the code. The expression in line 11

```
static_assert(Numeric<T>);
```

checks whether `multiply`'s argument type (`T`) satisfies the **`Numeric`** **concept**'s requirements. If it does, `Numeric<T>` evaluates to `true`, and the compiler continues compiling the code. When we call `multiply` from lines 17 and 18, `Numeric<T>` evaluates to `true` because the argument types `int` and `double` both satisfy the concept `Numeric`.

However, when we call `multiply` in line 19 with `string`-object literals, `Numeric<T>` in line 11 evaluates to `false` because a `string` is not `Numeric`. The output window shows the error messages produced by **Clang C++**. We highlighted the key messages in bold and added blank lines for readability.

For the **`Numeric`** **concept**, the compiler tells you

```
note: because 'std::basic_string<char>' does not satisfy 'Numeric'
```

The messages also provide more detail, saying that

```
note: because 'std::basic_string<char>' does not satisfy 'integral'
```

and that

```
note: and 'std::basic_string<char>' does not satisfy 'floating_point'
```

A `static_assert` declaration optionally may specify as a second argument a string to include in the error message for a `false` assertion.

15.8 Creating a Custom Algorithm

We saw in Chapters 13 and 14 that the standard library is divided into containers, iterators and algorithms. We showed algorithms operating on container elements via iterators. You can take advantage of this architecture to **define custom algorithms capable of operating on any container that supports your algorithm's iterator requirements**. Figure 15.10 defines a constrained `average` algorithm in which the argument must satisfy the custom `NumericInputRange` concept in lines 14–17, which we describe after the figure.

```
1   // fig15_10.cpp
2   // A custom algorithm to calculate the average of
3   // a numeric input range's elements.
4   #include <algorithm>
5   #include <array>
6   #include <concepts>
7   #include <iostream>
8   #include <iterator>
9   #include <list>
10  #include <ranges>
11  #include <vector>
12
13  // concept for an input range containing integer or floating-point values
14  template<typename T>
15  concept NumericInputRange = std::ranges::input_range<T> &&
16     (std::integral<typename T::value_type> ||
17     std::floating_point<typename T::value_type>);
18
19  // calculate the average of a NumericInputRange's elements
20  auto average(NumericInputRange auto const& range) {
21     long double total{0};
22
23  for (auto i{range.begin()}; i != range.end(); ++i) {
24     total += *i; // dereference iterator and add value to total
25  }
26
27     // divide total by the number of elements in range
28     return total / std::ranges::distance(range);
29  }
30
31  int main() {
32     std::ostream_iterator<int> outputInt(std::cout, " ");
33     const std::array ints{1, 2, 3, 4, 5};
34     std::cout << "array ints: ";
35     std::ranges::copy(ints, outputInt);
36     std::cout << "\naverage of ints: " << average(ints);
```

```
37
38      std::ostream_iterator<double> outputDouble(std::cout, " ");
39      const std::vector doubles{10.1, 20.2, 35.3};
40      std::cout << "\n\nvector doubles: ";
41      std::ranges::copy(doubles, outputDouble);
42      std::cout << "\naverage of doubles: " << average(doubles);
43
44      std::ostream_iterator<long double> outputLongDouble(std::cout, " ");
45      const std::list longDoubles{10.1L, 20.2L, 35.3L};
46      std::cout << "\n\nlist longDoubles: ";
47      std::ranges::copy(longDoubles, outputLongDouble);
48      std::cout << "\naverage of longDoubles: " << average(longDoubles)
49      << "\n";
50  }
```

```
array ints: 1 2 3 4 5
average of ints: 3

vector doubles: 10.1 20.2 35.3
average of doubles: 21.8667

list longDoubles: 10.1 20.2 35.3
average of doubles: 21.8667
```

Fig. 15.10 A custom algorithm to calculate the average of a numeric input range.

Concepts Custom `NumericInputRange` Concept

The **custom `NumericInputRange` concept** (lines 14–17) checks

- whether a type satisfies the `input_range` concept (line 15), so the argument supports at least **input iterators** for reading its elements, and
- whether a range's elements satisfy the `std::integral` or `std::floating_point` concepts (lines 16–17), so they can be used in calculations.

Recall from Section 13.2.1 that **each standard container has a nested `value_type`**, which indicates the container's element type. We use this to check whether the element type satisfies this concept's requirements. In lines 16 and 17, the notation

```
typename T::value_type
```

indicates that **`T::value_type`** is an **alias for the range's element type**.

Custom `average` Algorithm

Lines 20–29 define `average` as an **abbreviated function template**. We constrained its `range` parameter with our **custom concept `NumericInputRange`**, so this algorithm can operate on any **`input_range`** of numeric values. To ensure that `average` can support any built-in numeric type, we use a `long double` (line 21) to store the sum of the elements.

Lines 23–25 iterate through the `range` from its **`begin` iterator** up to, but not including, its **`end` iterator** and add each element's value to the total. Then line 28 divides the `total` by the range's number of elements, as determined by the **`std::ranges::distance` algorithm**.

Using Our `average` Algorithm on Standard Library Containers

To show that our custom `average` algorithm can process various standard library containers, lines 33, 39 and 45 define an `array` of `int`s, a `vector` of `double`s and a `list` of `long double`s, respectively. Lines 36, 42 and 48 call `average` with each of these containers to calculate the averages of their elements.

15.9 Creating a Custom Container and Iterators

CG Chapter 11 introduced our `MyArray` class to demonstrate special member functions and operator overloading. The C++ Core Guidelines recommend using templates to implement any class representing a container of values or range of values.[71] So, here, we'll define `MyArray` as a class template and enhance it using various standard library conventions.[72] We'll also define **custom iterators** to use `MyArray` objects with various standard library algorithms.

71. C++ Core Guidelines, "T.3: Use Templates to Express Containers and Ranges." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-cont.

72. Jonathan Boccara, "Make Your Containers Follow the Conventions of the STL," April 24, 2018. Accessed February 2, 2022. https://www.fluentcpp.com/2018/04/24/following-conventions-stl/.

Our goal here is to give you a sense of what's involved in creating standard-library-like containers. Achieving full standard library compatibility and backward compatibility with prior C++ language versions involves many **conditional compilation directives** beyond this book's scope. If you'd like to build reusable standard-library-like containers, study the code provided by the compiler vendors, and check out the research sources cited in our footnotes. Figure 15.11 defines our `MyArray` class template and its custom iterators. We broke the figure into several parts for discussion.

Single Header File

SE One change you'll notice from Chapter 11's `MyArray` class is that **the entire class and its custom iterators are defined in a single header**, which is typical of class templates. **The compiler needs the complete definition where the template is used to instantiate it.** Also, defining member functions inside the class template definition simplifies the syntax, as you do not need `template` headers for each member function.

`MyArray` Supports Bidirectional Iterator

We modeled this example after the standard library's `array` class template, which uses a compile-time allocated, fixed-size, **built-in array**. As discussed in Section 13.3, pointers into such arrays satisfy all the requirements of random-access iterators. However, for this example, we'll implement **custom bidirectional iterators** using class templates. The iterator architecture we use was inspired by the **Microsoft open-source C++ standard library** `array` implementation,[73] which **defines two iterator classes**:

73. "`array` Standard Header." Latest commit (i.e., when the file was last updated) February 24, 2021. Accessed February 2, 2022. https://github.com/microsoft/STL/blob/main/stl/inc/array.

- one for **iterators that manipulate `const` objects**, and
- one for **iterators that manipulate non-`const` objects**.

You can view Microsoft's implementation at

```
https://github.com/microsoft/STL/blob/main/stl/inc/array
```

We defined the following custom iterator classes:

- Our `ConstIterator` class represents a **read-only bidirectional iterator**.
- Our `Iterator` class represents a **read/write bidirectional iterator**.

The **GNU** and **Clang** `array` implementations simply use pointers for their `array` iterators. You can see their implementations at

```
https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/
   include/std/array
```

and

```
https://github.com/llvm/llvm-project/blob/main/libcxx/include/array
```

Why We Implemented Bidirectional Rather Than Random-Access Iterators

Recall from our introduction to iterators in Section 13.3 that various levels of iterators are supported by standard library containers. The most powerful are **contiguous iterators**, which are **random-access iterators** for containers that guarantee contiguous memory. Most standard library algorithms operate on ranges of container elements. **Only 12 require random-access iterators**—`shuffle` and 11 sorting-related algorithms—and none require **contiguous iterators**. `MyArray`**'s bidirectional iterators enable most standard library algorithms to process** `MyArray` **objects**. Random-access iterators have many additional requirements[74] shown in the table below. As an exercise, you could enhance `MyArray`'s custom iterators with these capabilities to make them **random-access iterators**.

74. C++ Standard, "23.3.4.13 Concept `random_access_iterator`." Accessed February 2, 2022. https://timsong-cpp.github.io/cppwp/n4861/iterator.concept.random.access.

Random-access iterator operation	Description
`p += i`	Increment the iterator `p` by `i` positions.
`p -= i`	Decrement the iterator `p` by `i` positions.
`p + i` *or* `i + p`	Result is an iterator positioned at `p` incremented by `i` positions.
`p - i`	Result is an iterator positioned at `p` decremented by `i` positions.
`p - p1`	Calculate the distance (that is, number of elements) between two elements in the same container.
`p[i]`	Return a reference to the element offset from `p` by `i` positions

Random-access iterator operation	Description
`p < p1`	Return `true` if iterator `p` is less than iterator `p1` (that is, iterator `p` is before iterator `p1` in the container); otherwise, return `false`.
`p <= p1`	Return `true` if iterator `p` is less than or equal to iterator `p1` (that is, iterator `p` is before or at the same location as iterator `p1` in the container); otherwise, return `false`.
`p > p1`	Return `true` if iterator `p` is greater than iterator `p1` (that is, iterator `p` is after iterator `p1` in the container); otherwise, return `false`.
`p >= p1`	Return `true` if iterator `p` is greater than or equal to iterator `p1` (that is, iterator `p` is after or at the same location as iterator `p1` in the container); otherwise, return `false`.

Basic Iterator Requirements

All iterators must support:[75,76,77]

75. "C++ Named Requirements: LegacyIterator." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/named_req/Iterator.
76. Triangles, "Writing a Custom Iterator in Modern C++," December 19, 2020. Accessed February 2, 2022. https://internalpointers.com/post/writing-custom-iterators-modern-cpp.
77. David Gorski, "Custom STL Compatible Iterators," March 2, 2019. Accessed February 2, 2022. https://davidgorski.ca/posts/stl-iterators/.

- default construction,
- copy construction,
- copy assignment,
- destruction and
- swapping.

We implement our iterator classes using the "**Rule of Zero**" (Section 11.6.6), letting the compiler generate the **copy constructor**, **copy assignment operator**, **move constructor**, **move assignment operator** and **destructor** special member functions.

15.9.1 Class Template `ConstIterator`

Lines 15–77 of Fig. 15.11 define the **class template** `ConstIterator`, which class `MyArray` uses to create **read-only iterators**. The template has one type parameter `T` (line 15), representing a `MyArray`'s element type. Each `ConstIterator` points to an element of that type.

```
1    // Fig. 15.11: MyArray.h
2    // Class template MyArray with custom iterators implemented
```

```
3    // by class templates ConstIterator and Iterator
4    #pragma once
5    #include <algorithm>
6    #include <compare>
7    #include <initializer_list>
8    #include <iostream>
9    #include <iterator>
10   #include <memory>
11   #include <stdexcept>
12   #include <utility>
13
14   // class template ConstIterator for a MyArray const iterator
15   template <typename T>
16   class ConstIterator {
17   public:
18     // public iterator nested type names
19     using iterator_category = std::bidirectional_iterator_tag;
20     using difference_type = std::ptrdiff_t;
21     using value_type = T;
22     using pointer = const value_type*;
23     using reference = const value_type&;
24
25     // default constructor
26     ConstIterator() = default;
27
28     // initialize a ConstIterator with a pointer into a MyArray
29     ConstIterator(pointer p) : m_ptr{p} {}
30
31     // OPERATIONS ALL ITERATORS MUST PROVIDE
32     // increment the iterator to the next element and
33     // return a reference to the iterator
34     ConstIterator& operator++() noexcept {
35        ++m_ptr;
36        return *this;
37     }
38
39     // increment the iterator to the next element and
40     // return the iterator before the increment
41     ConstIterator operator++(int) noexcept {
42     ConstIterator temp{*this};
43        ++(*this);
44       return temp;
45    }
46
47    // OPERATIONS INPUT ITERATORS MUST PROVIDE
48    // return a const reference to the element m_ptr points to
49    reference operator*() const noexcept {return *m_ptr;}
50
51    // return a const pointer to the element m_ptr points to
52    pointer operator->() const noexcept {return m_ptr;}
53
54    // <=> operator automatically supports equality/relational operators.
55    // Only == and != are needed for bidirectional iterators.
56    // This implementation would support the <, <=, > and >= required
57    // by random-access iterators.
```

```
58     auto operator<=>(const ConstIterator& other) const = default;
59
60     // OPERATIONS BIDIRECTIONAL ITERATORS MUST PROVIDE
61     // decrement the iterator to the previous element and
62     // return a reference to the iterator
63     ConstIterator& operator--() noexcept {
64        --m_ptr;
65        return *this;
66     }
67
68     // decrement the iterator to the previous element and
69     // return the iterator before the decrement
70     ConstIterator operator--(int) noexcept {
71        ConstIterator temp{*this};
72        --(*this);
73       return temp;
74    }
75  private:
76    pointer m_ptr{nullptr};
77  };
78
```

Fig. 15.11 Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—class `ConstIterator`.

Standard Iterator Nested Type Names

Lines 19–23 define type aliases for the **nested type names that the C++ standard library expects in iterator classes:**[78]

78. C++ Standard, "23.3.2.3 Iterator Traits." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/iterators#iterator.traits.

- 20 **`iterator_category`**: The iterator's category (Section 13.3.2), specified here as the type **`std::bidirectional_iterator_tag`** from header `<iterator>`. This "`tag`" type indicates **bidirectional iterators**. The standard library algorithms use type traits and C++20 concepts to confirm that a container's iterators have the correct category for use with each algorithm.
- **`difference_type`**: The result type of subtracting one `ConstIterator` from another—**`std::ptrdiff_t`** is the result type for pointer subtraction.
- **`value_type`**: The element type to which a `ConstIterator` points.
- **`pointer`**: The type of a pointer to a `const` object of the `value_type`. The `Const-Iterator`'s data member (line 76) is declared with this type.
- **`reference`**: The type of a reference to a `const` object of the `value_type`.

Constructors

Class `ConstIterator` provides a no-argument `default`ed constructor (line 26) that initializes a `ConstIterator`'s `m_ptr` member using its in-class initializer (`nullptr`; line 76), and a constructor that initializes a `ConstIterator` from a `pointer` to an element (line 29). **The class's copy and move constructors are autogenerated.**

++ Operators

Lines 34–37 and 41–45 define the preincrement and postincrement operators required by all iterators. The preincrement operator aims the iterator's `m_ptr` member at the next element and

returns a reference to the incremented iterator. The postincrement operator aims the iterator's `m_ptr` member at the next element and returns a copy of the iterator before the increment.

Overloaded * and -> Operators

Semantically, iterators are like pointers, so they must overload the `*` and `->` operators (lines 49 and 52). The overloaded `operator*` dereferences `m_ptr` to access the element the iterator currently points to and returns a `reference` to that element. The overloaded `operator->` returns `m_ptr` as a `pointer` to that element.

Bidirectional Iterator Comparisons

Bidirectional iterators must be comparable with `==` and `!=`. Here we used the **compiler-generated three-way comparison operator** `<=>` (line 58) to support `ConstIterator` comparisons. If you enhance our iterator classes to make them random-access iterators, this implementation also enables comparisons with the operators <, <=, > and >=, as required by random-access iterators.

--Operators

Lines 63–66 and 70–74 define the predecrement and postdecrement operators required by bidirectional iterators. These operators work like the `++` operators but aim the `m_ptr` member at the previous element.

15.9.2 Class Template Iterator

Lines 81–137 of Fig. 15.11 define the **class template** `Iterator`, which class `MyArray` uses to create **read/write iterators**. The template has one type parameter `T` (line 81), representing a `MyArray`'s element type. Class `Iterator` inherits from class `ConstIterator<T>` (line 82).

```
79   // class template Iterator for a MyArray non-const iterator;
80   // redefines several inherited operators to return non-const results
81   template <typename T>
82   class Iterator : public ConstIterator<T> {
83   public:
84      // public iterator nested type names
85      using iterator_category = std::bidirectional_iterator_tag;
86      using difference_type = std::ptrdiff_t;
87      using value_type = T;
88      using pointer = value_type*;
89      using reference = value_type&;
90
91      // inherit ConstIterator constructors
92      using ConstIterator<T>::ConstIterator;
93
94     // OPERATIONS ALL ITERATORS MUST PROVIDE
95     // increment the iterator to the next element and
96     // return a reference to the iterator
97     Iterator& operator++() noexcept {
98        ConstIterator<T>::operator++(); // call base-class version
99        return *this;
100    }
101
102     // increment the iterator to the next element and
```

```
103     // return the iterator before the increment
104     Iterator operator++(int) noexcept {
105        Iterator temp{*this};
106        ConstIterator<T>::operator++(); // call base-class version
107        return temp;
108      }
109
110      // OPERATIONS INPUT ITERATORS MUST PROVIDE
111      // return a reference to the element m_ptr points to; this
112      // operator returns a non-const reference for output iterator support
113     reference operator*() const noexcept {
114        return const_cast<reference>(ConstIterator<T>::operator*());
115     }
116
117     // return a pointer to the element m_ptr points to
118     pointer operator->() const noexcept {
119        return const_cast<pointer>(ConstIterator<T>::operator->());
120      }
121
122     // OPERATIONS BIDIRECTIONAL ITERATORS MUST PROVIDE
123     // decrement the iterator to the previous element and
124     // return a reference to the iterator
125     Iterator& operator--() noexcept {
126     ConstIterator<T>::operator--(); // call base-class version
127       return *this;
128   }
129
130     // decrement the iterator to the previous element and
131     // return the iterator before the decrement
132     Iterator operator--(int) noexcept {
133        Iterator temp{*this};
134        ConstIterator<T>::operator--(); // call base-class version
135        return temp;
136   }
137 };
138
```

Fig. 15.11 Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—class `Iterator`.

Standard Iterator Nested Type Names

Lines 85–89 define **type aliases for the nested type names that the C++ standard library expects in iterator classes**. Class template `Iterator` defines **read/write iterators**, so lines 88–89 define the `pointer` and `reference` **type aliases** without `const`. Pointers and references of these types can be used to write new values into elements.

Constructors

`ConstIterator`'s constructors know how to initialize `Iterator`'s inherited `m_ptr` member, so line 92 simply inherits the base class's constructors.

++ and -- Operators

Lines 97–100, 104–108, 125–128 and 132–136 define the preincrement, postincrement, predecrement and postdecrement operators. Each simply calls **`ConstIterator`**'s corresponding version. The prefix

operators `return` a reference to the updated `Iterator`. The postfix operators return a copy of the `Iterator` before the increment or decrement.

Overloaded * and -> Operators

Lines 113–115 and 118–120 overload the `*` and `->` operators. Each calls **`ConstIterator`**'s version. Those versions return a pointer or reference that views the `value_type` as `const`. Since `Iterator`s should allow both reading and writing element values, lines 114 and 119 use **`const_cast`** to **cast away the `const`-ness** (that is, remove the **`const`**) of the pointer or reference returned by the base-class overloaded operators.

15.9.3 Class Template `MyArray`

Lines 141–211 of Fig. 15.11 define our simplified `MyArray` class template. We removed some overloaded operators and various special member functions presented in Chapter 11 to focus on the **container** and its **iterators**. To mimic the `std::array` class template, we define `MyArray` as an **aggregate type** (Section 9.21), which requires all non-`static` data to be `public`. So, we defined `MyArray` using a `struct`, which has `public` members by default. Also, the new `MyArray` class template's data is stored as a fixed-size built-in array (line 210) allocated at compile-time, rather than using dynamic memory allocation.

```
139 // class template MyArray contains a fixed-size T[SIZE] array;
140 // MyArray is an aggregate type with public data, like std::array
141 template <typename T, size_t SIZE>
142 struct MyArray {
143    // type names used in standard library containers
144    using value_type = T;
145    using size_type = size_t;
146    using difference_type = ptrdiff_t;
147    using pointer = value_type*;
148    using const_pointer = const value_type*;
149    using reference = value_type&;
150    using const_reference = const value_type&;
151
152    // iterator type names used in standard library containers
153    using iterator = Iterator<T>;
154    using const_iterator = ConstIterator<T>;
155    using reverse_iterator = std::reverse_iterator<iterator>;
156    using const_reverse_iterator = std::reverse_iterator<const_iterator>;
157
158    // Rule of Zero: MyArray's special member functions are autogenerated
159
160    constexpr size_type size() const noexcept {return SIZE;} // return size
161
162    // member functions that return iterators
163    iterator begin() {return iterator{&m_data[0]};}
164    iterator end() {return iterator{&m_data[0] + size()};}
165    const_iterator begin() const {return const_iterator{&m_data[0]};}
166    const_iterator end() const {
167       return const_iterator{&m_data[0] + size()};
168    }
169    const_iterator cbegin() const {return begin();}
```

```
170     const_iterator cend() const {return end();}
171
172     // member functions that return reverse iterators
173     reverse_iterator rbegin() {return reverse_iterator{end()};}
174     reverse_iterator rend() {return reverse_iterator{begin()};}
175     const_reverse_iterator rbegin() const {
176        return const_reverse_iterator{end()};
177     }
178     const_reverse_iterator rend() const {
179        return const_reverse_iterator{begin()};
180     }
181     const_reverse_iterator crbegin() const {return rbegin();}
182     const_reverse_iterator crend() const {return rend();}
183
184     // autogenerated three-way comparison operator
185     auto operator<=>(const MyArray& t) const noexcept = default;
186
187     // overloaded subscript operator for non-const MyArrays;
188     // reference return creates a modifiable lvalue
189     T& operator[](size_type index) {
190       // check for index out-of-range error
191       if (index >= size()) {
192          throw std::out_of_range{"Index out of range"};
193       }
194
195       return m_data[index]; // reference return
196     }
197
198   // overloaded subscript operator for const MyArrays;
199   // const reference return creates a non-modifiable lvalue
200   const T& operator[](size_type index) const {
201     // check for subscript out-of-range error
202     if (index >= size()) {
203       throw std::out_of_range{"Index out of range"};
204     }
205
206     return m_data[index]; // returns copy of this element
207   }
208
209     // like std::array the data is public to make this an aggregate type
210     T m_data[SIZE]; // built-in array of type T with SIZE elements
211 };
212
```

Fig. 15.11 Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—class `MyArray`.

`MyArray`'s template Heamder

The `template` header (line 141) indicates that `MyArray` is a **class template**. The header specifies two parameters:

- `T` represents the `MyArray`'s element type.
- `SIZE` is a **non-type template parameter** that's treated as a compile-time constant. We use `SIZE` to represent the `MyArray`'s number of elements.

Line 210 creates a built-in array of type `T` containing `SIZE` elements. Though we do not do so here, non-type template parameters can have **default arguments**.

Standard Container Nested Type Names

Lines 144–156 define type aliases for the nested type names that the C++ standard library expects in container classes. The types for **reverse iterators** are specific to containers with at least **bidirectional iterators**:[79]

79. C++ Standard, "Table 73: Container Requirements." Accessed February 2, 2022. https://timsongcpp.github.io/cppwp/n4861/container.requirements#tab:container.req.

- **`value_type`**: The container's element type (`T`).
- **`size_type`**: The type representing the container's number of elements.
- **`difference_type`**: The result type when subtracting iterators.
- **`pointer`**: The type of a pointer to a `value_type` object.
- **`const_pointer`**: The type of a pointer to a `const value_type` object.
- **`reference`**: The type of a reference to a `value_type` object.
- **`const_reference`**: The type of a reference to a `const value_type` object.
- **`iterator`**: `MyArray`'s read/write iterator type (`Iterator<T>`).
- **`const_iterator`**: `MyArray`'s read-only iterator type (`ConstIterator<T>`).
- **`reverse_iterator`**: `MyArray`'s read/write iterator type for iterating backward from the end of a `MyArray`. The **iterator adapter** **`std::reverse_iterator`** creates a reverse-iterator type from its iterator type argument, which must be at least bidirectional.
- **`const_reverse_iterator`**: `MyArray`'s read-only iterator type for moving backward from the end of the `MyArray`—`std::reverse_iterator` creates a const reverse-iterator type from its iterator type argument, which must be at least bidirectional.

`MyArray` Member Functions That Return Iterators

Lines 163–182 define the `MyArray` member functions that return `MyArray`'s various kinds of iterators and reverse iterators. The key functions are `begin` and `end` in lines 163–168:

- **`begin`** (line 163) returns an `iterator` pointing to the `MyArray`'s first element.
- **`end`** (line 164) returns an `iterator` pointing to one past the `MyArray`'s last element.
- **`begin`** (line 165) is a `const`-qualified overload that returns a `const_iterator` pointing to the `MyArray`'s first element.
- **`end`** (line 166–168) is a `const`-qualified overload that returns a `const_iterator` pointing to one past the `MyArray`'s last element.

The other member functions that return iterators call these `begin` and `end` functions:

- **`cbegin`** and **`cend`** (lines 169–170) call the versions of `begin` and `end` that return `const` iterators.
- **`rbegin`** and **`rend`** (lines 173–174) produce `reverse_iterator`s based on the `iterator`s returned by the **non-`const` versions of `begin` and `end`**.
- **`rbegin`** and **`rend`** (lines 175–180) are overloads that produce `const_reverse_iterator`s based on the `const_iterator`s returned by the **`const` versions of `begin` and `end`**.
- **`crbegin`** and **`crend`** (lines 181–182) return the `const_reverse_iterator`s from calling the `const` versions of `rbegin` and `rend`.

MyArray Overloaded Operators

Lines 185, 189–196 and 200–207 define `MyArray`'s overloaded operators. Though we won't use it in this example, the **compiler-generated three-way comparison operator** (line 185) enables you to compare entire `MyArray` objects of the same element type and size. The overloaded `operator[]` member functions are identical to those in Chapter 11's `MyArray` class, but we now call the `size` member function (lines 191 and 202) when determining whether an `index` is outside a `MyArray`'s bounds.

15.9.4 MyArray Deduction Guide for Braced Initialization

As you've seen, you can initialize a `std::array` using **class template argument deduction (CTAD)**. For example, when the compiler sees the statement:

```
std::array ints{1, 2, 3, 4, 5};
```

it infers that the `array`'s element type is `int` because all the initializers are `int`s, and it counts the initializers to determine the `array`'s size. `MyArray` does not define a constructor that can receive any number of arguments. However, we can define a **deduction guide**[80] (lines 214–215) that shows the compiler how to deduce a `MyArray`'s type from a braced initializer. Then the compiler can use **aggregate initialization** (e.g., line 15 in Fig. 15.12) to place the initializers into our `MyArray` aggregate type's built-in array data member.

80. "Class Template Argument Deduction." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/class_template_argument_deduction.

```
213 // deduction guide to enable MyArrays to be brace initialized
214 template<typename T, std::same_as<T>... Us>
215 MyArray(T first, Us... rest) -> MyArray<T, 1 + sizeof...(Us)>;
```

Fig. 15.11 Class template `MyArray` with custom iterators implemented by class templates `ConstIterator` and `Iterator`—`MyArray` deduction guide.

A deduction guide is a **template**. This deduction guide's `template` header uses **variadic template syntax (`...`)**, which we discuss extensively in Section 15.12. Line 214 indicates that the compiler is looking for an initializer list that

- contains at least one initializer, specified by `typename T`, and
- may contain any number of additional initializers of the same type as `T`, specified by `std::same_as<T>... Us`>. The `...` indicates a **parameter pack**. A **parameter pack's** name (`Us`) is typically plural because it can represent multiple items.

To the left of the `->` in line 215, you specify what looks like the beginning of a `MyArray` constructor that receives its first argument in the `T` parameter named `first` and all other arguments in the `Us...` parameter named `rest`. To the right of the `->`, you tell the compiler that when it sees a `MyArray` initialized using **class template argument deduction**, it should deduce that we want to create a `MyArray` with elements of type `T` and with its size specified by

```
1 + sizeof...(Us)
```

In this expression,

- `1` is the initializer list's minimum number of initializers and
- `sizeof...(Us)` uses C++11's compile-time **`sizeof...` operator** to determine the additional number of initializers the compiler placed in the parameter pack `Us`.

Our deduction guide is based on those provided by GNU and Clang for their `std::array` implementations. You can view GNU's and Clang's deduction guides in their respective `<array>` headers:

```
https://github.com/gcc-mirror/gcc/blob/master/libstdc%2B%2B-v3/
```

```
   include/std/array
```
and

```
https://github.com/llvm/llvm-project/blob/main/libcxx/include/array
```

15.9.5 Using `MyArray` and Its Custom Iterators with `std::ranges` Algorithms

Figure 15.12 creates three `MyArray`s that store `int`s, `double`s and `string`s, respectively, then uses them with various `std::ranges` algorithms that require **input**, **output**, **forward** or **bidirectional iterators**. We also use a **range-based `for` statement** to iterate through a `MyArray`. `MyArray` has **bidirectional iterators**, so we also could use it with our custom `average` algorithm in Fig. 15.10, which required only **input iterators**.

```
1   // fig15_12.cpp
2   // Using MyArray with range-based for and with
3   // C++ standard library algorithms.
4   #include <iostream>
5   #include <iterator>
6   #include "MyArray.h"
7
8   int main() {
9       std::ostream_iterator<int> outputInt{std::cout, " "};
10      std::ostream_iterator<double> outputDouble{std::cout, " "};
11      std::ostream_iterator<std::string> outputString{std::cout, " "};
12
13      std::cout << "Displaying MyArrays with std::ranges::copy, "
14         << "which requires input iterators:\n";
15      MyArray ints{1, 2, 3, 4, 5, 6, 7, 8};
16      std::cout << "ints: ";
17      std::ranges::copy(ints, outputInt);
18
19      MyArray doubles{1.1, 2.2, 3.3, 4.4, 5.5};
20      std::cout << "\ndoubles: ";
21      std::ranges::copy(doubles, outputDouble);
22
23      using namespace std::string_literals; // for string object literals
24      MyArray strings{"red"s, "orange"s, "yellow"s};
25      std::cout << "\nstrings: ";
26      std::ranges::copy(strings, outputString);
27
28      std::cout << "\n\nDisplaying a MyArray with a range-based for "
29         << "statement, which requires input iterators:\n";
30      for (const auto& item : doubles) {
31         std::cout << item << " ";
32      }
33
34      std::cout << "\n\nCopying a MyArray with std::ranges::copy, "
35        << "which requires an input range and an output iterator:\n";
36      MyArray<std::string, strings.size()> strings2{};
37      std::ranges::copy(strings, strings2.begin());
```

```
38      std::cout << "strings2 after copying from strings: ";
39      std::ranges::copy(strings2, outputString);
40
41      std::cout << "\n\nFinding min and max elements in a MyArray "
42        << "with std::ranges::minmax_element, which requires "
43        << "a forward range:\n";
44       auto [min, max] {std::ranges::minmax_element(strings)};
45      std::cout << "min and max elements of strings are: "
46        << *min << ", " << *max;
47
48      std::cout << "\n\nReversing a MyArray with std::ranges::reverse, "
49        << "which requires a bidirectional range:\n";
50      std::ranges::reverse(ints);
51      std::cout << "ints after reversing elements: ";
52      std::ranges::copy(ints, outputInt);
53      std::cout << "\n";
54   }
```

```
Displaying MyArrays with std::ranges::copy, which requires input iterators:
ints: 1 2 3 4 5 6 7 8
doubles: 1.1 2.2 3.3 4.4 5.5
strings: red orange yellow

Displaying a MyArray with a range-based for statement, which requires input iterators:
1.1 2.2 3.3 4.4 5.5

Copying a MyArray with std::ranges::copy, which requires an input range and output
iterator:
 strings2 after copying from strings: red orange yellow

Finding min and max elements in a MyArray with std::ranges::minmax_element, which
requires a forward range:
min and max elements of strings are: orange, yellow

Reversing a MyArray with std::ranges::reverse, which requires a bidirectional range:
ints after reversing elements: 8 7 6 5 4 3 2 1
```

Fig. 15.12 Using `MyArray` with range-based `for` and with C++ standard library algorithms.

Creating `MyArrays` and Displaying Them with `std::ranges::copy`

In lines 13–26, we create three `MyArrays` (lines 15, 19 and 24), using **class template argument deduction** to infer their element types and sizes. Lines 17, 21 and 26 display each `MyArray`'s contents using **`std::ranges::copy`**. This algorithm's first argument is an **`input_range`**, which requires the range to have **input iterators**. `MyArrays` are compatible with this algorithm because they have more powerful **bidirectional iterators**.

Displaying a `MyArray` with a Range-Based `for` Statement

17 Lines 30–32 display the `MyArray doubles` using a range-based `for`, which requires only **input iterators**, so it **works with any iterable object**, including `MyArrays`. The `doubles MyArray` is not a `const` object, so `MyArray`'s read/write iterators are used. **If you have a non-`const` object that you want to treat as `const`, you can create a `const` view of the non-`const` object by passing it to C++17's `std::as_const` function**. For example, this example's range-based `for` does not modify `doubles`' elements, so we could have written line 30 as

```
for (auto& item : std::as_const(doubles)) {
```

In this case, `item`'s type will be inferred as a reference to a `const double` element.

Copying a `MyArray` with `std::ranges::copy`

Line 36 creates a new `MyArray` into which we'll copy the `MyArray strings`' elements. Line 37 uses **`std::ranges::copy`** to copy `strings`' elements into `strings2`. The algorithm uses an **input iterator** to read each element from `strings` and an **output iterator** to specify where to write the element into `strings2`. `MyArray`'s **non-`const` bidirectional iterators** support both reading (input) and writing (output), so one `MyArray` can be copied into another of the same type with **`std::ranges::copy`**.

Finding the Minimum and Maximum Elements in a `MyArray` with `std::ranges::minmax_element`

Line 44 uses the **`std::ranges::minmax_element`** algorithm to get iterators pointing to the elements containing a `MyArray`'s minimum and maximum values. This algorithm requires a **`forward_range`**, which provides **forward iterators**. `MyArray`'s **bidirectional iterators** are more powerful than **forward iterators**, so the algorithm can operate on a `MyArray`.

Reversing a `MyArray` with `std::ranges::reverse`

Line 50 reverses `MyArray ints`' elements using the **`std::ranges::reverse`** algorithm, which requires a **`bidirectional_range`** with **bidirectional iterators**. These are the exact iterators provided by `MyArray`, so the algorithm can operate on a `MyArray`.

Attempting to Use `MyArray` with `std::ranges::sort`

Err `MyArray`'s iterators do not support all the features of **random-access iterators**, so we **cannot pass a `MyArray` to algorithms that require them**. To prove this, we wrote a short program containing only the following statements that create a `MyArray` of `int`s then attempt to sort it with **`std::ranges::sort`**, which requires **random-access iterators**:

```
MyArray integers{10, 2, 33, 4, 7, 1, 80};
std::ranges::sort(integers);
```

The **Clang** compiler produced the error messages in the output window below. We highlighted key messages in bold and added some blank lines for readability. The messages clearly indicate that the code does not compile

```
because 'MyArray<int, 7> &' does not satisfy 'random_access_range'
```

and

```
because 'iterator_t<MyArray<int, 7> &>' (aka 'Iterator<int>') does
not satisfy 'random_access_iterator'
```

```
test.cpp:9:4: error: no matching function for call to object of type 'const
std::ranges::__sort_fn'
   std::ranges::sort(integers);
```

```
   ^~~~~~~~~~~~~~~~~

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/rang-
es_algo.h:2032:7: note: candidate template ignored: constraints not satisfied
[with _Range = MyArray<int, 7> &, _Comp = std::ranges::less, _Proj =
std::identity]
     operator()(_Range&& __r, _Comp __comp = {}, _Proj __proj = {}) const
     ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/rang-
es_algo.h:2028:14: note: because 'MyArray<int, 7> &' does not satisfy 'ran-
dom_access_range'
     template<random_access_range _Range,
              ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/
range_access.h:934:37: note: because 'iterator_t<MyArray<int, 7> &>' (aka 'It-
erator<int>') does not satisfy 'random_access_iterator'
      = bidirectional_range<_Tp> && random_access_iterator<iterator_t<_Tp>>;
                                    ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/iter-
ator_concepts.h:614:10: note: because 'derived_from<__detail::__iter_con-
cept<Iterator<int> >, std::random_access_iterator_tag>' evaluated to false
   && derived_from<__detail::__iter_concept<_Iter>,
      ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:67:28: note: because '__is_base_of(std::random_access_iterator_tag,
std::bidirectional_iterator_tag)' evaluated to false
    concept derived_from = __is_base_of(_Base, _Derived)
                           ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/bits/rang-
es_algo.h:2019:7: note: candidate function template not viable: requires at
least 2 arguments, but 1 was provided
      operator()(_Iter __first, _Sent __last,
      ^

1 error generated.
```

15.10 Default Arguments for Template Type Parameters

A type parameter also can specify a **default type argument**. For example, the C++ standard's `stack` **container adapter** class template begins with:

```
template <class T, class Container = deque<T>>
```

which specifies that a `stack` uses a `deque` by default to store the `stack`'s elements of type `T`. When the compiler sees the declaration

```
stack<int> values;
```

it **instantiates class-template `stack` for type `int`** and uses the resulting specialization to instantiate the object named `values`. The `stack`'s elements are stored in a **`deque<int>`**.

11 **Default type parameters** must be the rightmost (trailing) parameters in a template's type-parameter list. When you instantiate a template with two or more default arguments, if an omitted argument is not the rightmost, all type parameters to the right of it also must be omitted. **C++11 added the ability to use default type arguments for template type parameters in function templates.**

15.11 Variable Templates

14 17 You've instantiated function templates and class templates to define groups of related functions and classes, respectively. C++14 added **variable templates**, which define groups of related variables. You used several **predefined variable templates** in Fig. 15.6's type traits demonstration. C++17 added convenient **variable templates for accessing each type trait's `value` member**. For example, the **variable template**:

```
is_integral_v<T>
```

is defined in the standard as

```
template<class T>
inline constexpr bool is_arithmetic_v = is_arithmetic<T>::value;
```

17 Err SE This variable template defines an **`inline bool`** variable that evaluates to a **compile-time constant** (indicated by `constexpr`). C++17 added `inline` variables to better support header-only libraries, which can be included in multiple source-code files (i.e., translation units) within the same application.[81] When you define a regular variable in a header, including that header more than once in an application results in multiple definition errors for that variable. On the other hand, **identical `inline` variable definitions are allowed in separate translation units within the same application.**[82]

81. Alex Pomeranz, "6.8—Global Constants and Inline Variables," January 3, 2020. Accessed February 2, 2022. https://www.learncpp.com/cpp-tutorial/global-constants-and-inline-variables/.

82. "`inline` Specifier." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/inline.

15.12 Variadic Templates and Fold Expressions

11 SE Before C++11, each class template or function template had a fixed number of template parameters. Defining a class or function template with different numbers of template parameters required a separate template definition for each case. C++11 **variadic templates accept any number of arguments**. They simplify template programming because you can provide one variadic function template rather than many overloaded ones with different numbers of parameters.

15.12.1 `tuple` Variadic Class Template

11 C++11's `tuple` class (from header **`<tuple>`**) is a **variadic-class-template generalization of class template `pair`** (introduced in Section 13.9). A **`tuple`** is a collection of related values, possibly of mixed types. Figure 15.13 demonstrates several `tuple` capabilities.

```cpp
// fig15_13.cpp
// Manipulating tuples.
#include <fmt/format.h>
#include <iostream>
#include <string>
#include <tuple>

// type alias for a tuple representing a hardware part's inventory
using Part = std::tuple<int, std::string, int, double>;

// return a part's inventory tuple
Part getInventory(int partNumber) {
    using namespace std::string_literals; // for string object literals

    switch (partNumber) {
    case 1:
        return {1, "Hammer"s, 32, 9.95}; // return a Part tuple
    case 2:
        return {2, "Screwdriver"s, 106, 6.99}; // return a Part tuple
    default:
        return {0, "INVALID PART"s, 0, 0.0}; // return a Part tuple
  }
}

  int main() {
    // display the hardware part inventory
  for (int i{1}; i <= 2; ++i) {
    // unpack the returned tuple into four variables;
    // variables' types are inferred from the tuple's element values
  auto [partNumber, partName, quantity, price] {getInventory(i)};

  std::cout << fmt::format("{}: {}, {}: {}, {}: {}, {}: {:.2f}\n",
     "Part number", partNumber, "Tool", partName,
     "Quantity", quantity, "Price", price);
   }

  std::cout << "\nAccessing a tuple's elements by index number:\n";
  auto hammer{getInventory(1)};
  std::cout << fmt::format("{}: {}, {}: {}, {}: {}, {}: {:.2f}\n",
     "Part number", std::get<0>(hammer), "Tool", std::get<1>(hammer),
     "Quantity", std::get<2>(hammer), "Price", std::get<3>(hammer));

  std::cout << fmt::format("A Part tuple has {} elements\n",
     std::tuple_size<Part>{}); // get the tuple size
}
```

```
Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
Part number: 2, Tool: Screwdriver, Quantity: 106, Price: 6.99

Accessing a tuple's elements by index number:
Part number: 1, Tool: Hammer, Quantity: 32, Price: 9.95
A Part tuple has 4 elements
```

Fig. 15.13 Manipulating tuples.

Line 9's `using` declaration (introduced in Section 10.13) creates a **type alias** named `Part` for a `tuple` representing a hardware part's inventory. Each `Part` consists of

- an `int` part number,
- a `string` part name,
- an `int` quantity and
- a `double` price.

In a `tuple` declaration, every template type parameter corresponds to a value at the same position in the `tuple`. **The number of `tuple` elements always matches the number of type parameters.** We use the `Part` type alias to simplify the rest of the code.

Packing a `tuple`

Creating a `tuple` object is called **packing a `tuple`**. Lines 12–23 define a `getInventory` function that receives a part number and returns a `Part` tuple. The function packs `Part` `tuple`s by returning an **initializer list** (lines 17, 19 and 21) containing four elements (an `int`, a `string`, an `int` and a `double`), which the compiler uses to initialize the returned `Part std::tuple` object. A `tuple`'s size is fixed once you create it.

Creating a `tuple` with `std::make_tuple`

You also can pack a `tuple` with the `<tuple>` header's **`make_tuple` function**, which infers a `tuple`'s type parameters from the function's arguments. For example, you could create the `tuple`s in lines 17, 19 and 21 with `make_tuple` calls like

```
std::make_tuple(1, "Hammer"s, 32, 9.95)
```

If we had used this approach, `getInventory`'s return type could be specified as **`auto` to infer the return type from `make_tuple`'s result**.

Unpacking a `tuple` with Structured Bindings

17 You can **unpack a `tuple`** to access its elements using **C++17 structured bindings** (Section 14.4.5). Line 30 unpacks a `Part`'s members into variables, which we display in lines 32–34.

Using `get<index>` to Obtain a `tuple` Member by Index

Class template `pair` contains public members `first` and `second` for accessing a `pair`'s two members. Each `tuple` you create can have any number of elements, so `tuple`s do not have similarly named data members. Instead, the `<tuple>` header provides the function template **`get< index>(tupleObject)`**, which returns a reference to the *tupleObject*'s member at the specified *index*. The first member has index `0`. Line 38 gets a tuple for the hammer inventory, then lines 39–41 access each `tuple` element by `index`.

C++14 Using `get<type>` to Obtain a `tuple` Member By Type

14 C++14 added a `get` overload that gets a `tuple` member of a specific type, provided that the tuple contains **only one member of that type**. For example, the following statement gets the `hammer` `tuple`'s `string` member:

```
auto partName{get<std::string>(hammer)};
```

However, the statement

```
auto partNumber{get<int>(hammer)};
```

Err would generate a compilation error because the call is **ambiguous**—the `hammer tuple` contains two `int` members for its part number and quantity.

Other `tuple` Features

The following table shows several other **`tuple` class template features**. For the `tuple` class template's complete details, see

```
https://en.cppreference.com/w/cpp/utility/tuple/tuple
```

For other `tuple`-related utilities defined in the `<tuple>` header, see

```
https://en.cppreference.com/w/cpp/utility/tuple
```

Other **tuple** class template features	Description
comparisons	Tuples that contain the same number of members can be compared to one another using the relational and equality operators (assuming their elements must support the appropriate operators).
default constructor	Creates a `tuple` in which each member is **value initialized**. Primitive type values are set to 0 or the equivalent of 0. Objects of class types are initialized with their default constructors.
copy constructor	Copies a `tuple`'s elements into a new `tuple` of the same type.
move constructor	Moves a `tuple`'s elements into a new `tuple` of the same type.
copy assignment	Uses the assignment operator (`=`) to copy `tuple` elements in the right operand into a `tuple` of the same type in the left operand.
move assignment	Uses the assignment operator (`=`) to move `tuple` elements in the right operand into a `tuple` of the same type in the left operand.

15.12.2 Variadic Function Templates and an Intro to C++17 Fold Expressions

Variadic function templates enable you to define functions that can receive any number of arguments. Figure 15.14 uses **variadic function templates** to sum one or more arguments. This example assumes the arguments can be operands to the `+` operator and have the same type. We show two ways to process the variadic parameters:

- using **compile-time recursion** (which was required before C++17), and
- using a **C++17 fold expression** to eliminate the recursion.

Section 15.12.7 shows how to **test whether all the arguments have the same type**.

```
// fig15_14.cpp
// Variadic function templates.
#include <fmt/format.h>
#include <iostream>
#include <string>

// base-case function for one argument
template <typename T>
auto sum(T item) {
  return item;
}

// recursively add one or more arguments
template <typename FirstItem, typename... RemainingItems>
auto sum(FirstItem first, RemainingItems... theRest) {
   return first + sum(theRest...); // expand parameter pack for next call
}

// add one or more arguments with a fold expression
template <typename FirstItem, typename... RemainingItems>
auto foldingSum(FirstItem first, RemainingItems... theRest) {
   return (first + ... + theRest); // expand the parameter
}

int main() {
    using namespace std::literals;

    std::cout << "Recursive variadic function template sum:"
        << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n\n",
               "sum(1): ", sum(1), "sum(1, 2): ", sum(1, 2),
               "sum(1, 2, 3): ", sum(1, 2, 3),
               "sum(\"s\"s, \"u\"s, \"m\"s): ", sum("s"s, "u"s, "m"s));

    std::cout << "Variadic function template foldingSum:"
        << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n",
               "sum(1): ", foldingSum(1), "sum(1, 2): ", foldingSum(1, 2),
               "sum(1, 2, 3): ", foldingSum(1, 2, 3),
               "sum(\"s\"s, \"u\"s, \"m\"s): ",
               foldingSum("s"s, "u"s, "m"s));
}
```

```
Recursive variadic function template sum:
sum(1): 1
```

```
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum

Variadic function template foldingSum:
sum(1): 1
sum(1, 2): 3
sum(1, 2, 3): 6
sum("s"s, "u"s, "m"s): sum
```

Fig. 15.14 Variadic function templates.

Compile-Time Recursion

Lines 8–11 and 14–17 define overloaded function templates named `sum` that use **compile-time recursion** to process **variadic parameter packs**. The function template `sum` with one template parameter (lines 8–11) represents the **recursion's base case**, in which `sum` receives only one argument and returns it. The **recursive function template** `sum` (lines 14–17) specifies two type parameters:

- The type parameter `FirstItem` represents the first function argument.
- The type parameter `RemainingItems` represents all the other arguments passed to the function.

The `typename...` introduces a variadic template's **parameter pack** representing **any number of arguments**. In the function's parameter list (line 15), the variable-length parameter must appear last and is denoted with `...` after its type (`RemainingItems`). Note that the `...` position is different in the template header and the function parameter list. The return statement adds the `first` argument to the result of the **recursive call**

```
sum(theRest...)
```

The expression `theRest...` is a **parameter-pack expansion**—the compiler turns the parameter pack's elements into a comma-delimited list. We'll say more about this in a moment.

Calling `sum` with One Argument

When line 30 calls

```
sum(1)
```

which has one argument, the compiler invokes `sum`'s one-argument version (lines 8–11). If we have only one argument, the sum is simply the argument's value (in this case, `1`).

Calling `sum` with Two Arguments

When line 30 calls

```
sum(1, 2)
```

the compiler invokes `sum`'s **variadic version** (lines 14–17), which receives `1` in the parameter `first` and `2` in the **parameter pack** `theRest`. The function then adds `1` and the result of calling `sum` with the **parameter-pack expansion (`...`)**. The parameter pack contains only `2`, so this call becomes `sum(2)`, invoking `sum`'s one-argument version, ending the **recursion**. So, the final result is `3` (i.e., `1 + 2`).

Though `sum`'s **variadic version** looks like traditional **recursion**, the compiler generates separate template instantiations for each **recursive call**. So, the `sum(1, 2)` call becomes

```
sum(1, sum(2))
```

or more precisely

```
sum<int, int>(1, sum<int>(2))
```

Perf The compiler has all the values used in the calculation, so it can perform the calculations and inline them in the program as compile-time constants, **eliminating execution-time function-call overhead.**[83]

83. "Template Metaprogramming—Compile-Time Code Optimization." Wikipedia. Wikimedia Foundation. Accessed February 2, 2022. https://en.wikipedia.org/wiki/Template_metaprogramming#Compile-time_code_optimization.

Calling `sum` with Three Arguments

When line 31 calls

```
sum(1, 2, 3)
```

the compiler again invokes `sum`'s **variadic version** (lines 14–17):

- In this initial call, parameter `first` receives `1` and the **parameter pack** `theRest` receives `2` and `3`. The function adds `1` and the result of calling `sum` with the **parameter-pack expansion**, producing the call `sum(2, 3)`.
- In the call `sum(2, 3)`, parameter `first` receives `2` and the **parameter pack** `theRest` receives `3`. The function then adds `2` and the result of calling `sum` with the **parameter-pack expansion**, producing the call `sum(3)`.
- The call `sum(3)` invokes `sum`'s one-argument version (the **base case**) with `3`, which returns 3. At this point, the `sum(2, 3)` call's body becomes `2 + 3` (that is `5`), and the `sum(1, 2, 3)` call's body becomes `1 + 5`, producing the final result `6`.

Effectively, the original call became

```
sum(1, sum(2, sum(3)))
```

where the compiler knows the value of the innermost call, `sum(3)`, and can determine the results of the other calls.

Calling `sum` with Three `string` Objects

Line 32's `sum` call

```
sum("s"s, "u"s, "m"s)
```

receives three `string`-object literals. Recall that `+` for `string`s performs **string concatenation**, so the original call effectively becomes

```
sum("s"s, sum("u"s, sum("m"s)))
```

producing the string `"sum"`.

C++17 Fold Expressions

Fold expressions provide a convenient notation for repeatedly applying a binary operator to all the elements in a **variadic template's parameter pack**.[84,85,86] Fold expressions are often used to **reduce the values in a parameter pack to a single value**. They also can **apply an operation to every object in a parameter pack**, such as calling a member function or displaying the pack's elements with `cout` (as we'll do in Section 15.12.6).

84. Jonathan Boccara, "C++ Fold Expressions 101," March 12, 2021. Accessed February 2, 2022. https://www.fluentcpp.com/2021/03/12/cpp-fold-expressions/.
85. Jonathan Boccara, "What C++ Fold Expressions Can Bring to Your Code," March 19, 2021. Accessed February 2, 2022. https://www.fluentcpp.com/2021/03/19/what-c-fold-expressions-can-bring-to-your-code/.
86. Jonathan Muller, "Nifty Fold Expression Tricks," May 5, 2020. Accessed February 2, 2022. https://www.foonathan.net/2020/05/fold-tricks/.

Lines 20–23 use a **binary left fold** (line 22)

```
(first + ... + theRest)
```

which sums `first` and the zero or more arguments in the parameter pack `theRest`. **Fold expressions must be parenthesized. A binary-left-fold expression has two binary operators, which must be the same**—in this case, the addition operator (`+`). The argument to the left of the first operator is the expression's **initial value**. The `...` expands the parameter pack to the right of the second operator, separating each parameter in the parameter pack from the next with the binary operator. So, in line 37, the function call

```
foldingSum(1, 2, 3)
```

the binary-left-fold expression expands to

```
((1 + 2) + 3)
```

If the parameter pack is empty, the value of the binary-left-fold expression is the initial value—in this case, the value of `first`.

The C++ Core Guidelines recommend using **variadic function templates** for arguments of **mixed types**[87] and `initializer_list`s for functions that receive variable numbers of arguments of the **same type**.[88] However, **fold expressions cannot be applied to `initializer_list`s.**

87. C++ Core Guidelines, "T.100: Use Variadic Templates When You Need a Function That Takes a Variable Number of Arguments of a Variety of Types." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic.
88. C++ Core Guidelines, "T.103: Don't Use Variadic Templates for Homogeneous Argument Lists." Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-variadic-not.

15.12.3 Types of Fold Expressions

In the following descriptions,

- *pack* represents a parameter pack,
- *op* represents one of the 32 binary operators you can use in fold expressions[89] and
- *initialValue* represents a starting value for a binary fold expression. For example, a sum might start with 0, and a product might start with 1.

89. "Fold Expression." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/language/fold.

There are **four fold-expression types**—the parentheses are required in each:

- A **unary left fold** has one binary operator with the parameter pack expansion (`...`) as the **left operand** and the *pack* as the right operand:
(`...` *op pack*)
- A **unary right fold** has one binary operator with the parameter pack expansion (`...`) as the **right operand** and the *pack* as the left operand:
(*pack op* `...`)
- A **binary left fold** has two binary operators, which must be the same. The *initial-Value* is to the **left of the first operator**, the parameter pack expansion (`...`) is between the operators, and *pack* is to the right of the second operator:
(*initialValue op* `...` *op pack*)
- A **binary right fold** has two binary operators, which must be the same. The *initialValue* is to the **right of the second operator**, the parameter pack expansion (`...`) is between the operators, and *pack* is to the left of the first operator:
(*pack op* `...` *op initialValue*)

15.12.4 How Unary Fold Expressions Apply Their Operators

The key difference between left- and right-fold expressions is the order in which they apply their operators. **Left folds group left-to-right and right folds group right-to-left**. Depending on the operator, the grouping can produce different results. Figure 15.15 demonstrates unary-left-fold and unary-right-fold operations using the addition (`+`) and subtraction (`-`) operators:

- Lines 6–9 define `unaryLeftAdd`, which uses a **unary left fold** to add the items in its parameter pack.
- Lines 11–14 define `unaryRightAdd`, which uses a **unary right fold** to add the items in its parameter pack.
- Lines 16–19 define `unaryLeftSubtract`, which uses a **unary left fold** to subtract the items in its parameter pack.
- Lines 21–24 define `unaryRightSubtract`, which uses a **unary right fold** to subtract the items in its parameter pack.

```
1  // fig15_15.cpp
2  // Unary fold expressions.
3  #include <fmt/format.h>
4  #include <iostream>
5
6  template <typename... Items>
7  auto unaryLeftAdd(Items... items) {
```

```
 8       return (... + items); // unary left fold
 9   }
10
11   template <typename... Items>
12   auto unaryRightAdd(Items... items) {
13       return (items + ...); // unary right fold
14   }
15
16   template <typename... Items>
17   auto unaryLeftSubtract(Items... items) {
18       return (... - items); // unary left fold
19   }
20
21   template <typename... Items>
22   auto unaryRightSubtract(Items... items) {
23       return (items - ...); // unary right fold
24   }
25
26   int main() {
27       std::cout << "Unary left and right fold with addition:"
28          << fmt::format("\n{}{}\n{}{}\n\n",
29               "unaryLeftAdd(1, 2, 3, 4): ", unaryLeftAdd(1, 2, 3, 4),
30               "unaryRightAdd(1, 2, 3, 4): ", unaryRightAdd(1, 2, 3, 4));
31
32       std::cout << "Unary left and right fold with subtraction:"
33          << fmt::format("\n{}{}\n{}{}\n",
34                "unaryLeftSubtract(1, 2, 3, 4): ",
35                unaryLeftSubtract(1, 2, 3, 4),
36               "unaryRightSubtract(1, 2, 3, 4): ",
37               unaryRightSubtract(1, 2, 3, 4));
38   }
```

```
Unary left and right fold with addition:
unaryLeftAdd(1, 2, 3, 4): 10
unaryRightAdd(1, 2, 3, 4): 10

Unary left and right fold with subtraction:
unaryLeftSubtract(1, 2, 3, 4): -8
unaryRightSubtract(1, 2, 3, 4): -2
```

Fig. 15.15 Unary fold expressions.

Unary Left and Right Folds for Addition

Consider the line 29 call

```
unaryLeftAdd(1, 2, 3, 4)
```

in which the **parameter pack** contains `1`, `2`, `3` and `4`. This function performs a **unary left fold** using the `+` operator, which calculates

```
(((1 + 2) + 3) + 4)
```

Similarly, the line 30 call

```
unaryRightAdd(1, 2, 3, 4)
```

performs a **unary right fold**, which calculates

```
(1 + (2 + (3 + 4)))
```

The order in which + is applied does not matter, so both expressions produce the same value (`10`) in this case.

Unary Left and Right Folds for Subtraction

The order in which some operators are applied can produce different results. Consider the line 35 call

```
unaryLeftSubtract(1, 2, 3, 4)
```

This function performs a **unary left fold** using the `-` operator, which calculates

```
(((1 - 2) - 3) - 4)
```

producing `-8`. On the other hand, the line 37 call

```
unaryRightSubtract(1, 2, 3, 4)
```

performs a **unary right fold** using the `-` operator, which calculates

```
(1 - (2 - (3 - 4)))
```

producing `-2`.

Parameter Packs in Unary Fold Expressions Must Not Be Empty

Err **Unary-fold expressions must be applied only to parameter packs with at least one element**; otherwise, a compilation error occurs. The only exceptions to this are for the binary operators `&&`, `||` and comma (`,`):

- An `&&` operation with an **empty parameter pack** evaluates to `true`.
- An `||` operation with an **empty parameter pack** evaluates to `false`.
- Any operation performed with the **comma operator** on an **empty parameter pack** evaluates to `void()`, which simply means no operation is performed. We show a fold expression using the comma operator in Section 15.12.6.

15.12.5 How Binary-Fold Expressions Apply Their Operators

If an **empty parameter pack** is possible, you can use a **binary left fold** or **binary right fold**. Each requires an initial value. If the **parameter pack** is empty, the initial value is used as the fold expression's value. Figure 15.16 demonstrates **binary left folds** and **binary right folds** for addition and subtraction:

- Lines 6–9 define `binaryLeftAdd`, which uses a **binary left fold** to add the items in its parameter pack starting with the initial value `0`.

- Lines 11–14 define `binaryRightAdd`, which uses a **binary right fold** to add the items in its parameter pack starting with the initial value `0`.
- Lines 16–19 define `binaryLeftSubtract`, which uses a **binary left fold** to subtract the items in its parameter pack starting with the initial value `0`.
- Lines 21–24 define `binaryRightSubtract`, which uses a **binary right fold** to subtract the items in its parameter pack starting with the initial value `0`.

Once again, note that the fold expressions' grouping for addition does not matter, but for subtraction leads to different results.

```
1   // fig15_16.cpp
2   // Binary fold expressions.
3   #include <fmt/format.h>
4   #include <iostream>
5
6   template <typename... Items>
7   auto binaryLeftAdd(Items... items) {
8      return (0 + ... + items); // binary left fold
9   }
10
11  template <typename... Items>
12  auto binaryRightAdd(Items... items) {
13    return (items + ... + 0); // binary right fold
14  }
15
16  template <typename... Items>
17  auto binaryLeftSubtract(Items... items) {
18    return (0 - ... - items); // binary left fold
19  }
20
21  template <typename... Items>
22  auto binaryRightSubtract(Items... items) {
23    return (items - ... - 0); // binary right fold
24  }
25
26  int main() {
27     std::cout << "Binary left and right fold with addition:"
28        << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n\n",
29              "binaryLeftAdd(): ", binaryLeftAdd(),
30              "binaryLeftAdd(1, 2, 3, 4): ", binaryLeftAdd(1, 2, 3, 4),
31              "binaryRightAdd(): ", binaryRightAdd(),
32              "binaryRightAdd(1, 2, 3, 4): ", binaryRightAdd(1, 2, 3, 4));
33
34     std::cout << "Binary left and right fold with subtraction:"
35        << fmt::format("\n{}{}\n{}{}\n{}{}\n{}{}\n",
36               "binaryLeftSubtract(): ", binaryLeftSubtract(),
37              "binaryLeftSubtract(1, 2, 3, 4): ",
38              binaryLeftSubtract(1, 2, 3, 4),
39              "binaryRightSubtract(): ", binaryRightSubtract(),
40              "binaryRightSubtract(1, 2, 3, 4): ",
41              binaryRightSubtract(1, 2, 3, 4));
42    }
```

```
Binary left and right fold with addition:
binaryLeftAdd(): 0
binaryLeftAdd(1, 2, 3, 4): 10
binaryRightAdd(): 0
binaryRightAdd(1, 2, 3, 4): 10

Binary left and right fold with subtraction:
binaryLeftSubtract(): 0
binaryLeftSubtract(1, 2, 3, 4): -10
binaryRightSubtract(): 0
binaryRightSubtract(1, 2, 3, 4): -2
```

Fig. 15.16 Binary fold expressions.

15.12.6 Using the Comma Operator to Repeatedly Perform an Operation

The **comma operator** evaluates the expression to its left, then the expression to its right. The value of a comma-operator expression is the value of the rightmost expression. You can **combine the comma operator with fold expressions to repeatedly perform tasks for every item in a parameter pack**. Figure 15.17's `printItems` function displays every item in its parameter pack (`items`) on a line by itself. Line 7 executes the expression on the left of the comma operator

```
(std::cout << items << "\n")
```

once for each item in the parameter pack `items`.

```
1   // fig15_17.cpp
2   // Repeating a task using the comma operator and fold expressions.
3   #include <iostream>
4
5   template <typename... Items>
6   void printItems(Items... items) {
7       ((std::cout << items << "\n"), ...); // unary right fold
8   }
9
10   int main() {
11       std::cout << "printItems(1, 2.2, \"hello\"):\n";
12       printItems(1, 2.2, "hello");
13   }
```

```
printItems(1, 2.2, "hello"):
1
2.2
hello
```

Fig. 15.17 Repeating a task using the comma operator and fold expressions.

15.12.7 Constraining Parameter Pack Elements to the Same Type

When using **fold expressions**, there may be cases in which you'd like every element to have the **same type**. For example, you might want a variadic function template to sum its arguments and produce a result of the same type as the arguments. Figure 15.18 uses the **predefined concept** `std::same_as` to check that all the elements of a parameter pack have the same type as another type argument.

```cpp
// fig15_18.cpp
// Constraining a variadic-function-template parameter pack to
// elements of the same type.
#include <concepts>
#include <iostream>
#include <string>

template <typename T, typename... Us>
concept SameTypeElements = (std::same_as<T, Us> && ...);

// add one or more arguments with a fold expression
template <typename FirstItem, typename... RemainingItems>
   requires SameTypeElements<FirstItem, RemainingItems...>
auto foldingSum(FirstItem first, RemainingItems... theRest) {
   return (first + ... + theRest); // expand the parameter
}

int main() {
   using namespace std::literals;

   foldingSum(1, 2, 3); // valid: all are int values
   foldingSum("s"s, "u"s, "m"s); // valid: all are std::string objects
   foldingSum(1, 2.2, "hello"s); // error: three different types
}
```

```
fig15_18.cpp:23:4: error: no matching function for call to 'foldingSum'
   foldingSum(1, 2.2, "hello"s); // error: three different types
   ^~~~~~~~~~

fig15_18.cpp:14:6: note: candidate template ignored: constraints not satisfied
[with FirstItem = int, RemainingItems = <double, std::basic_string<char>>]
auto foldingSum(FirstItem first, RemainingItems... theRest) {
     ^

fig15_18.cpp:13:13: note: because 'SameTypeElements<int, double,
std::basic_string<char> >' evaluated to false
   requires SameTypeElements<FirstItem, RemainingItems...>
            ^
fig15_18.cpp:9:34: note: because 'std::same_as<int, double>' evaluated to
false
concept SameTypeElements = (std::same_as<T, Us> && ...);
                                 ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:63:19: note: because '__detail::__same_as<int, double>' evaluated to
false
      = __detail::__same_as<_Tp, _Up> && __detail::__same_as<_Up, _Tp>;
                  ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:57:27: note: because 'std::is_same_v<int, double>' evaluated to false
      concept __same_as = std::is_same_v<_Tp, _Up>;
                          ^
fig15_18.cpp:9:34: note: and 'std::same_as<int, std::basic_string<char> >'
evaluated to false
concept SameTypeElements = (std::same_as<T, Us> && ...);
```

```
                                  ^

/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:63:19: note: because '__detail::__same_as<int, std::basic_string<char>
>' evaluated to false
      = __detail::__same_as<_Tp, _Up> && __detail::__same_as<_Up, _Tp>;
                  ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/10/../../../../include/c++/10/con-
cepts:57:27: note: because 'std::is_same_v<int, std::basic_string<char> >'
evaluated to false
      concept __same_as = std::is_same_v<_Tp, _Up>;
                          ^

1 error generated.
```

Fig. 15.18 Constraining a variadic-function-template parameter pack to elements of the same type.

Concepts Custom Concept for a Variadic Template

Lines 8–9 define a custom concept `SameTypeElements` with two type parameters:

- the first represents a single type, and
- the second is a **variadic type parameter**.

The constraint

```
std::same_as<T, Us>
```

compares the type `T` to one type from the parameter pack `Us`. This constraint is in a fold expression, so the **parameter pack expansion (...)** applies `std::same_as<T, Us>` once for each type from the parameter pack `Us`, comparing every type in `Us` to the type `T`. If every item in the parameter pack `Us` is the same as `T`, then the constraint evaluates to `true`. Line 13 in the `foldingSum` function template applies our `SameTypeElements` concept to the function's type parameters.

Calling `foldingSum`

Err In `main`, lines 21 and 22 call `foldingSum` with three `int`s and three `string`s, respectively. Each call has three arguments of the same type, so these statements will successfully compile. However, line 23 attempts to call `foldingSum` with an `int`, a `double` and a `string`. In that call, type `T` in the `SameTypeElements` concept is `int`, and types `double` and `string` are placed in the parameter pack `Us`. Of course, both `double` and `string` are not `int`, so each use of the concept `std::same_as<T, Us>` fails. Figure 15.18 shows the **Clang C++** compiler error messages. We highlighted key messages in bold and added vertical spacing for readability. Clang indicates

```
error: no matching function for call to 'foldingSum'
```

and

```
note: candidate template ignored: constraints not satisfied
```

It also points out in its notes each individual `std::same_as` test that failed because a type was not the same as `int`:

```
note: because 'std::same_as<int, double>' evaluated to false

note: and 'std::same_as<int, std::basic_string<char> >' evaluated to
false
```

15.13 Template Metaprogramming

As we've mentioned, a goal of modern C++ is to do work at compile-time to enable the compiler to optimize your code for runtime performance. Much of this optimization work is done via **template metaprogramming (TMP)**, which enables the compiler to

- manipulate types,
- perform compile-time computations and
- generate optimized code.

The **concepts**, **concept-based overloading** and **type traits** capabilities we've introduced in this chapter all are crucial template-metaprogramming capabilities.

SE Template metaprogramming is complex. When properly used, it can help you improve the runtime performance of your programs. So it's important to be aware of template metaprogramming's powerful capabilities.

Our goal in this section is to present manageable template-metaprogramming examples. We provide many footnote citations containing resources for further study. We'll show examples demonstrating

- computing values at compile-time with metafunctions,
- computing values at compile-time with `constexpr` functions,
- using type traits at compile-time via the `constexpr if` statement to optimize runtime performance and
- manipulating types at compile-time with metafunctions.

15.13.1 C++ Templates Are Turing Complete

Todd Veldhuizen proved that C++ templates are **Turing complete**,[90,91] so anything that can be computed, can be computed at compile-time with C++ template metaprogramming. One of the first attempts to demonstrate such compile-time computation was presented in 1994 during C++'s early standardization efforts. Erwin Unruh wrote a template metaprogram to calculate the prime numbers less than 30.[92,93] The program did not compile, but the compiler's error messages included the prime-number calculation results. You can view the original program—which is no longer valid C++—and the original compiler error messages at

90. Todd Veldhuizen, "C++ Templates Are Turing Complete," 2003. Accessed February 2, 2022. https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.3670.
91. "Template Metaprogramming." Wikipedia. Wikimedia Foundation. Accessed February 2, 2022. https://en.wikipedia.org/wiki/Template_metaprogramming.
92. Erwin Unruh, "Prime Number Computation," 1994. ANSI X3J16-94-0075/ISO WG21-4-62.

93. Rainer Grimm, “C++ Core Guidelines: Rules for Template Metaprogramming,” January 7, 2019. Accessed February 2, 2022. https://www.modernescpp.com/index.php/c-core-guidelines-rules-for-template-metaprogramming. Our thanks to Rainer Grimm for this blog post that pointed us to the historical note about Erwin Unruh.

```
http://www.erwin-unruh.de/primorig.html
```

Here are some selected lines from those error messages. The specializations of his class template `D` showing the first several prime numbers are highlighted in bold:

```
|    Type `enum{}´ can´t be converted to txpe `D<2>´ ("primes.cpp",L2/C25).
|    Type `enum{}´ can´t be converted to txpe `D<3>´ ("primes.cpp",L2/C25).
|    Type `enum{}´ can´t be converted to txpe `D<5>´ ("primes.cpp",L2/C25).
|    Type `enum{}´ can´t be converted to txpe `D<7>´ ("primes.cpp",L2/C25).
```

15.13.2 Computing Values at Compile-Time

Perf CG The goal of compile-time calculations is to optimize a program’s runtime performance.[94] Figure 15.19 uses template metaprogramming to calculate factorials at compile-time. First, we use a recursive factorial definition implemented with templates. Then, we show two compile-time `constexpr` functions using the iterative and recursive algorithms presented in Section 5.17. The C++ Core Guidelines recommend using `constexpr` functions to compute values at compile-time[95]—such functions use traditional C++ syntax.

94. “Template Metaprogramming.” Wikipedia.

95. C++ Core Guidelines, “T.123: Use `constexpr` Functions to Compute Values at Compile Time.” Accessed February 2, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rt-fct.

```
1   // fig15_19.cpp
2   // Calculating factorials at compile time.
3   #include <iostream>
4
5   // Factorial value metafunction calculates factorials recursively
6   template <int N>
7   struct Factorial {
8      static constexpr long long value{N * Factorial<N - 1>::value};
9   };
10
11  // Factorial specialization for the base case
12  template <>
13  struct Factorial<0> {
14     static constexpr long long value{1}; // 0! is 1
15  };
16
17  // constexpr compile-time recursive factorial
18  constexpr long long recursiveFactorial(int number) {
19     if (number <= 1) {
```

```
20          return 1; // base cases: 0! = 1 and 1! = 1
21       }
22       else { // recursion step
23          return number * recursiveFactorial(number - 1);
24       }
25    }
26
27    // constexpr compile-time iterative factorial
28    constexpr long long iterativeFactorial(int number) {
29       long long factorial{1}; // result for 0! and 1!
30
31       for (long long i{2}; i <= number; ++i) {
32          factorial *= i;
33       }
34
35       return factorial;
36    }
37
38    int main() {
39       // "calling" a value metafunction requires instantiating
40       // the template and accessing its static value member
41       std::cout << "CALCULATING FACTORIALS AT COMPILE TIME "
42          << "WITH A RECURSIVE METAFUNCTION"
43          << "\nFactorial(0): " << Factorial<0>::value
44          << "\nFactorial(1): " << Factorial<1>::value
45          << "\nFactorial(2): " << Factorial<2>::value
46          << "\nFactorial(3): " << Factorial<3>::value
47          << "\nFactorial(4): " << Factorial<4>::value
48          << "\nFactorial(5): " << Factorial<5>::value;
49
50       // calling the recursive constexpr function recursiveFactorial
51       std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE TIME "
52          << "WITH A RECURSIVE CONSTEXPR FUNCTION"
53          << "\nrecursiveFactorial(0): " << recursiveFactorial(0)
54          << "\nrecursiveFactorial(1): " << recursiveFactorial(1)
55          << "\nrecursiveFactorial(2): " << recursiveFactorial(2)
56          << "\nrecursiveFactorial(3): " << recursiveFactorial(3)
57          << "\nrecursiveFactorial(4): " << recursiveFactorial(4)
58          << "\nrecursiveFactorial(5): " << recursiveFactorial(5);
59
60      // calling the iterative constexpr function iterativeFactorial
61      std::cout << "\n\nCALCULATING FACTORIALS AT COMPILE TIME "
62          << "WITH AN ITERATIVE CONSTEXPR FUNCTION"
63          << "\niterativeFactorial(0): " << iterativeFactorial(0)
64          << "\niterativeFactorial(1): " << iterativeFactorial(1)
65          << "\niterativeFactorial(2): " << iterativeFactorial(2)
66          << "\niterativeFactorial(3): " << iterativeFactorial(3)
67          << "\niterativeFactorial(4): " << iterativeFactorial(4)
68          << "\niterativeFactorial(5): " << iterativeFactorial(5) << "\n";
69 }
```

```
CALCULATING FACTORIALS AT COMPILE TIME WITH A RECURSIVE METAFUNCTION
Factorial(0): 1
Factorial(1): 1
Factorial(2): 2
Factorial(3): 6
```

```
Factorial(4): 24
Factorial(5): 120

CALCULATING FACTORIALS AT COMPILE TIME WITH A RECURSIVE CONSTEXPR FUNCTION
recursiveFactorial(0): 1
recursiveFactorial(1): 1
recursiveFactorial(2): 2
recursiveFactorial(3): 6
recursiveFactorial(4): 24
recursiveFactorial(5): 120

CALCULATING FACTORIALS AT COMPILE TIME WITH AN ITERATIVE CONSTEXPR FUNCTION
iterativeFactorial(0): 1
iterativeFactorial(1): 1
iterativeFactorial(2): 2
iterativeFactorial(3): 6
iterativeFactorial(4): 24
iterativeFactorial(5): 120
```

Fig. 15.19 Calculating factorials at compile-time.

Metafunctions

You can perform compile-time calculations with **metafunctions**. Like the functions you've defined so far, metafunctions have arguments and return values, but their syntax is significantly different. **Metafunctions are implemented as class templates**—typically using `struct`s. A metafunction's arguments are the items used to specialize the class template, and its return value is a `public` class member. The type traits introduced in Section 15.5 are metafunctions.
There are two types of metafunctions:

- A **value metafunction** is a class template with a **`public static constexpr` data member typically named `value`**. The class template uses its template arguments to compute `value` at compile-time. This is how we implement factorial calculations.
- A **type metafunction** is a class template with a **nested `type` member**, typically defined as a **type alias**. The class template uses its template arguments to manipulate a type at compile-time. Section 15.13.4 presents type metafunctions.

The metafunction member names `value` and `type` are conventions[96] used throughout the C++ standard library and in metaprogramming in general.

96. Jody Hagins, "Template Metaprogramming: Type Traits (Part 1 of 2)," YouTube video, September 22, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=tiAVWcjIF6o.

`Factorial` Metafunction

Our `Factorial` metafunction (lines 6–9) is implemented using the recursive factorial definition

$$n! = n \cdot (n - 1)!$$

`Factorial` has one **non-type template parameter**—the `int` parameter `N` (line 6)—representing the metafunction's argument. Per the C++ standard's conventions for value meta-functions, line 8 defines a `public static constexpr` data member named `value`. Factorials grow quickly, so we declared the variable's type as `long long`. The constant `value` is initialized with the result of the expression

```
N * Factorial<N - 1>::value
```

which multiplies `N` by the result of "calling" the `Factorial` metafunction for `N - 1`.

"Calling" a Metafunction

You "call" a metafunction by instantiating its template, which probably feels weird to you. For example, to calculate the factorial of 3, you'd use

```
Factorial<3>::value
```

in which `3` is the **template argument** and `::value` is the "**return value.**" This expression causes the compiler to create a new type representing the factorial of 3. We'll say more about this momentarily.

`Factorial` Metafunction Specialization for the Base Case—`Factorial<0>`

For the recursive factorial calculation, 0! is the **base case**, which is defined to be 1. In **meta-function recursion**, you specify the base case with a **full template specialization** (lines 12–15). Such a specialization uses the notation `template <>` (line 12) to indicate that all the template's arguments will be specified explicitly in the angle brackets following the class name. The full template specialization `Factorial<0>` matches only `Factorial` calls with the argument `0`. Line 14 sets the specialization's `value` member to be `1`.[97]

97. Section 5.17 specified that 0 and 1 are both base cases for factorial calculations. To mimic that, we could implement a second full template specialization named `Factorial<1>`. As implemented, the `Factorial` metafunction handles `Factorial<1>` as `1 * Factorial<0>`.

How the Compiler Evaluates Metafunction `Factorial` for the Argument `0`

When the compiler encounters a metafunction call, such as **`Factorial<0>::value`** (line 43 in `main`), it must determine which `Factorial` class template to use. The template argument is the `int` value `0`. The class template `Factorial` in lines 6–9 can match any `int` value, and the full-template specialization `Factorial<0>` in lines 12–15 specifically matches only the value 0. **The compiler always chooses the most specialized template from multiple matching templates.** So, `Factorial<0>` matches the full template specialization, and `Factorial<0>::value` evaluates to `1`.

How the Compiler Evaluates Metafunction `Factorial` for the Argument `3`

Now, let's reconsider the metafunction call **`Factorial<3>::value`** (line 46 in `main`). The compiler generates each specialization needed to produce the final result at compile-time. When the compiler encounters **`Factorial<3>::value`**, it specializes the class template in lines 6–9 with `3` as the argument. In doing so, line 8 of the specialization becomes

```
3 * Factorial<2>::value
```

The compiler sees that to complete the **`Factorial<3>`** definition, it must specialize the template again for **`Factorial<2>`**. In that specialization, line 8 becomes

```
2 * Factorial<1>::value
```

Next, the compiler sees that to complete the **`Factorial<2>`** definition, it must specialize the template again for **`Factorial<1>`**. In that specialization, line 8 becomes

```
1 * Factorial<0>::value
```

The compiler knows that **`Factorial<0>::value`** is **`1`** from the full specialization in lines 12–15. This completes the `Factorial` specializations for **`Factorial<3>`**.

Now, the compiler knows everything it needs to complete the earlier `Factorial` specializations:

- **`Factorial<1>::value`** stores the result of `1 * 1`, which is `1`.
- **`Factorial<2>::value`** stores the result of `2 * 1`, which is `2`.
- **`Factorial<3>::value`** stores the result of `3 * 2`, which is `6`.

So the compiler can insert the final constant value `6` in place of the original metafunction call—**without any runtime overhead**. Each of the `Factorial` metafunction calls in lines 43–48 of `main` are evaluated similarly by the compiler, **enabling it to perform these calculations at compile-time.**

Functional Programming

None of what the compiler does during **template specialization** modifies variable values after they're initialized.[98] **There are no mutable variables in template metaprograms**. This is a hallmark of **functional programming**. All the values processed in this example are compile-time constants.

98. "Template Metaprogramming." Wikipedia. .

Using `constexpr` Functions to Perform Compile-Time Calculations

As you can see, using metafunctions to calculate values at compile-time is not as straightforward as using traditional runtime functions. C++ provides two options for compile-time evaluation of traditional functions:

- A function declared `constexpr` can be **called at compile-time or runtime**.
- 20 In C++20, you can declare a function `consteval` (rather than `constexpr`) to indicate that it can be **called only at compile-time** to produce a constant.

SE **Many computations performed with metafunctions are easier to implement using traditional functions that are declared `constexpr` or `consteval`. Such functions also are part of compile-time metaprogramming**. In fact, various members of the C++ standard committee prefer `constexpr`-based metaprogramming over template-based metaprogramming, which is "difficult to use, does not scale well and is basically equivalent to inventing a new language within C++."[99] They've proposed various `constexpr` enhancements to simplify other aspects of metaprogramming in future C++ versions.

99. Louis Dionne, "Metaprogramming By Design, Not By Accident," June 18, 2017. Accessed February 2, 2022. https://wg21.link/p0425r0.

Lines 18–25 and 28–36 define the `recursiveFactorial` and `iterativeFactorial` functions using the algorithms from Section 5.17 and traditional C++ syntax. Each function is declared `constexpr`, so the compiler can evaluate the function's result at compile-time. Lines 53–58 demonstrate `recursiveFactorial`, and lines 63–68 demonstrate `iterativeFactorial`.

15.13.3 Conditional Compilation with Template Metaprogramming and `constexpr if`

20 17 Another aspect of doing more at compile-time to optimize runtime execution is generating code that executes optimally at runtime. Section 15.6.4 showed the C++20 way to optimize runtime execution for a `customDistance` function template using **concept-based function overloading**. In that example, the compiler chose the correct overloaded function template to call based on the template parameter's type constraints. Figure 15.20 reimplements that example using a single `customDistance` function template, showing how you could implement similar functionality prior to concepts. Inside the function, a C++17 **compile-time `if` statement**—known as a **`constexpr if`**—uses **`std::is_base_of_v`** (the shorthand for accessing **type trait `std::is_base_of`**'s `value` member) to decide whether to generate the $O(1)$ or $O(n)$ distance calculation based on whether the function's iterator arguments are

- random-access iterators (or the derived iterator category contiguous iterators) that can be used to perform an $O(1)$ distance calculation or
- lesser iterators for which we'll perform the $O(n)$ distance calculation.

```
// fig15_20.cpp
// Implementing customDistance using template metaprogramming.
#include <array>
#include <iostream>
#include <iterator>
#include <list>
#include <ranges>
#include <type_traits>

// calculate the distance (number of items) between two iterators;
// requires at least input iterators
template <std::input_iterator Iterator>
auto customDistance(Iterator begin, Iterator end) {
   // for random-access iterators, subtract the iterators
   if constexpr (std::is_base_of_v<std::random_access_iterator_tag,
      Iterator::iterator_category>) {

      std::cout << "customDistance with random-access iterators\n";
      return end - begin; // O(1) operation for random-access iterators
   }
   else { // for all other iterators
      std::cout << "customDistance with non-random-access iterators\n";
      std::ptrdiff_t count{0};

      // increment from begin to end and count number of iterations;
      // O(n) operation for non-random-access iterators
      for (auto iter{begin}; iter != end; ++iter) {
         ++count;
      }

      return count;
   }
}

int main() {
   const std::array ints1{1, 2, 3, 4, 5}; // has random-access iterators
   const std::list ints2{1, 2, 3}; // has bidirectional iterators

   auto result1{customDistance(ints1.begin(), ints1.end())};
   std::cout << "ints1 number of elements: " << result1 << "\n";
   auto result2{customDistance(ints2.begin(), ints2.end())};
   std::cout << "ints2 number of elements: " << result2 << "\n";
}
```

```
customDistance with random-access iterators
ints1 number of elements: 5
customDistance with non-random-access iterators
ints2 number of elements: 3
```

Fig. 15.20 Implementing `customDistance` using template metaprogramming.

Lines 12–33 define function template `customDistance`. Since this function requires **at least input iterators** to perform its task, line 12 constrains the type parameter `Iterator` using the **concept** `std::input_iterator`. Though the **compile-time** `if` **statement** is known in the C++ standard as a `constexpr if`, it's written in code as `if constexpr` (line 15). This statement's condition must evaluate at compile-time to a `bool`.

Lines 15–16 use `std::is_base_of_v` to compare two types:

- `std::random_access_iterator_tag` and
- `Iterator::iterator_category`.

Recall that a standard iterator has a **nested type named** `iterator_category`, which designates the iterator's type using one of the following "`tag`" types from **header** `<iterator>`:

- `input_iterator_tag`
- `output_iterator_tag`
- `forward_iterator_tag`
- `bidirectional_iterator_tag`
- `random_access_iterator_tag`
- `contiguous_iterator_tag`

These are implemented as a class hierarchy.[100] In line 16, the expression

100. "`std::input_iterator_tag`, `std::output_iterator_tag`, `std::forward_iterator_tag`, `std::bidirectional_iterator_tag`, `std::random_access_iterator_tag`, `std::contiguous_iterator_tag`." Accessed February 2, 2022. https://en.cppreference.com/w/cpp/iterator/iterator_tags.

```
Iterator::iterator_category
```

gets the iterator tag from the argument's iterator type, which is `std::is_base_of_v`, then compares it to its first type argument

```
std::random_access_iterator_tag
```

to determine whether they're either the same type or `std::random_access_iterator_tag` is a base class of `Iterator::iterator_category`. The result is `true` or `false`. If `true`, the compiler instantiates the code in the `if constexpr` body (lines 18–19), which is optimized for **random-access iterators** to perform the **O(1)** calculation. Otherwise, the compiler instantiates the `else`'s body (lines 22–31), which uses the **O(n)** approach that works for all other iterator types.

The `g++` and `clang++` std::`array` implementations use pointers as iterators. These will not work with this `customDistance` implementation, whereas the concept-based overloading example in Section 15.6.4 does work with poiners as iterators.

15.13.4 Type Metafunctions

SE **Type metafunctions are frequently used to add attributes to and remove attributes from types at compile-time**. This is a more advanced template-metaprogramming technique used primarily by template class-library implementers. In Fig. 15.21, we'll add and remove type attributes using our own type metafunctions that mimic ones from the `<type_traits>` **header** so you can see how they might be implemented. We'll also use predefined ones. **Always prefer the** `<type_traits>` **header's predefined type traits rather than implementing your own.**

```cpp
// fig15_21.cpp
// Adding and removing type attributes with type metafunctions.
#include <fmt/format.h>
#include <iostream>
#include <type_traits>

// add const to a type T
template <typename T>
struct my_add_const {
  using type = const T;
};

// general case: no pointer in type, so set nested type variable to T
template <typename T>
struct my_remove_ptr {
   using type = T;
 };

// partial template specialization: T is a pointer type, so remove *
template <typename T>
struct my_remove_ptr<T*> {
   using type = T;
 };

int main() {
   std::cout << fmt::format("{}\n{}{}\n{}{}\n\n",
      "ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION",
      "std::is_same_v<const int, my_add_const<int>::type>: ",
      std::is_same_v<const int, my_add_const<int>::type>,
      "std::is_same_v<int* const, my_add_const<int*>::type>: ",
      std::is_same_v<int* const, my_add_const<int*>::type>);

  std::cout << fmt::format("{}\n{}{}\n{}{}\n\n",
      "REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION",
      "std::is_same_v<int, my_remove_ptr<int>::type>: ",
      std::is_same_v<int, my_remove_ptr<int>::type>,
      "std::is_same_v<int, my_remove_ptr<int*>::type>: ",
      std::is_same_v<int, my_remove_ptr<int*>::type>);

  std::cout << fmt::format("{}\n{}{}\n{}{}\n\n",
      "ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS",
      "std::is_same_v<int&, std::add_lvalue_reference<int>::type>: ",
      std::is_same_v<int&, std::add_lvalue_reference<int>::type>,
      "std::is_same_v<int&&, std::add_rvalue_reference<int>::type>: ",
      std::is_same_v<int&&, std::add_rvalue_reference<int>::type>);

  std::cout << fmt::format("{}\n{}{}\n{}{}\n{}{}\n",
      "REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS",
      "std::is_same_v<int, std::remove_reference<int>::type>: ",
      std::is_same_v<int, std::remove_reference<int>::type>,
      "std::is_same_v<int, std::remove_reference<int&>::type>: ",
      std::is_same_v<int, std::remove_reference<int&>::type>,
      "std::is_same_v<int, std::remove_reference<int&&>::type>: ",
      std::is_same_v<int, std::remove_reference<int&&>::type>);
  }
```

```
ADD CONST TO A TYPE VIA A CUSTOM TYPE METAFUNCTION
std::is_same_v<const int, my_add_const<int>::type>: true
std::is_same_v<int* const, my_add_const<int*>::type>: true

REMOVE POINTER FROM TYPES VIA A CUSTOM TYPE METAFUNCTION
std::is_same_v<int, my_remove_ptr<int>::type>: true
std::is_same_v<int, my_remove_ptr<int*>::type>: true

ADD REFERENCES TO TYPES USING STANDARD TYPE TRAITS
std::is_same_v<int&, std::add_lvalue_reference<int>::type>: true
std::is_same_v<int&&, std::add_rvalue_reference<int>::type>: true

REMOVE REFERENCES FROM TYPES USING STANDARD TYPE TRAITS
std::is_same_v<int, std::remove_reference<int>::type>: true
std::is_same_v<int, std::remove_reference<int&>::type>: true
std::is_same_v<int, std::remove_reference<int&&>::type>: true
```

Fig. 15.21 Compile-time type manipulation.

Adding `const` to a Type

Lines 8–11 implement a **type metafunction** named **`my_add_const`** that's a simplified version of the **`std::add_const` metafunction** from header **`<type_traits>`**. By convention, a **type metafunction must have a `public` member named `type`**. When **`my_add_const`** is instantiated with a type, the compiler defines the type alias in line 10, which precedes the type with **`const`**.

Lines 26–31 in **`main`** demonstrate the **`my_add_const` metafunction**. Line 29

```
my_add_const<int>::type
```

instantiates the template with the non-**`const`** type **`int`**. To confirm that **`const`** was added to **`int`**, we use **`std::is_same_v`** (the shorthand for accessing **`std::is_same`'s `value`** member) to compare the type **`const int`** to the type returned by the preceding expression. They are the same, so the result is **`true`**, as shown in the output. Similarly, line 31 shows that the **`my_add_const` metafunction** also works with pointer types.

Removing * from a Pointer Type

Lines 14–17 and 20–23 implement a **custom type metafunction named `my_remove_ptr`** that mimics the **`std::remove_pointer` metafunction** from the **`<type_traits>` header** to show how that metafunction could be implemented. This requires two metafunction class templates:

- The one in lines 14–17 handles the general case in which a type is not a pointer. Line 16's type alias defines the **`type`** member simply as **`T`**, which can be any type. For a type that does not include an **`*`** to indicate a pointer, this metafunction simply returns the type used to specialize the template.
- The metafunction class template in lines 20–23 matches only pointer types. For this purpose, we define a partial template specialization.[101] Unlike a **full template specialization** that begins with the **template header `template<>`**, a partial template specialization still specifies a **`template`** header (line 20) with one or more parameters, which it uses in the angle brackets following the class name (line 21). In this case, the partial specialization is that **`T`** must be a pointer—indicated with **`T*`**. To remove the **`*`** from the pointer type, the type alias in line 22 defines the **`type`** member simply as **`T`**.

101. Inbal Levi, "Exploration of C++20 Metaprogramming," September 29, 2020. Accessed February 2, 2022. https://www.youtube.com/watch?v=XgrjybKaIV8.

Lines 36 and 38 in `main` demonstrate our **custom `my_remove_ptr` type metafunctions**. Let's consider the compiler's matching process for invoking them.

Instantiating `my_remove_pointer` for Non-Pointer Types

The expression in line 36

```
my_remove_ptr<int>::type
```

instantiates the template with the non-pointer type `int`. The compiler must decide which definition of `my_remove_ptr` matches the expression. Since there is no `*` to indicate a pointer in the type `int`, this expression matches only the `my_remove_ptr` definition in lines 14–17, which sets its `type` member to the type argument `int`. To confirm this, we use `std::is_same_v` to compare type `int` to the type returned by the preceding expression. They are the same, so the result is `true`.

Instantiating `my_remove_pointer` for Pointer Types

The expression in line 38

```
my_remove_ptr<int*>::type
```

instantiates the template with the pointer type `int*`, which can match both definitions of `my_remove_const`:

- the first definition can match any type and
- the second can match any pointer type.

Again, the **compiler always chooses the most specific matching template**. In this case, the type `int*` matches the **partial template specialization** in lines 20–23:

- the `int` in the preceding expression matches `T` in line 21 and
- the `*` in the preceding expression matches the `*` in line 21, separating it from the type `int`—this is what enables the **partial template specialization** to remove the pointer (`*`) from the type.

To confirm that the `*` was removed, we use `std::is_same_v` to compare `int` to the type returned by the preceding expression. They are the same, so the result is `true`.

Adding *lvalue* and *rvalue* References to a Type

The **predefined type metafunctions** that modify types work similarly to `my_add_const` and `my_remove_ptr` type metafunctions. Lines 43 and 45 in `main` demonstrate the **predefined type metafunctions** `add_lvalue_reference` and `add_rvalue_reference`. Line 43 converts type `int` to type `int&`—an **lvalue reference type**. Line 45 converts type `int` to type `int&&`—an **rvalue reference type**. We use `std::is_same_v` to confirm both results.

Removing lvalue and rvalue References from a Reference Type

Lines 50, 52 and 54 test the `<type_traits>` **header**'s **predefined** `std::remove_reference` **metafunction**, which removes references from a type. We instantiate `std::remove_reference` with the types `int`, `int&` and `int&&`, respectively:

- non-reference types (like `int`) are returned as is,
- *lvalue* reference types have the `&` removed, and
- *rvalue* reference types have the `&&` removed.

To confirm this, we use `std::is_same_v` to compare `int` to the type returned by each expression in lines 50, 52 and 54. They are the same, so the result is `true` in each case.

15.14 Wrap-Up

In this chapter, we demonstrated the power of generic programming and compile-time polymorphism with templates and concepts. We used class templates to create related custom classes, distinguishing between the templates you write and template specializations the compiler generates from your code when you instantiate templates.

We introduced C++20's abbreviated function templates and templated lambdas. We used C++20 concepts to constrain template parameters and overload function templates based on their type requirements. Next, we introduced type traits and showed how they relate to C++20 concepts. We created our own custom concept and showed how to test it at compile-time with `static_assert`.

We demonstrated how to create a custom concept-constrained algorithm, then used it to manipulate objects of several C++ standard library container classes. Next, we rebuilt our class `MyArray` as a custom container class template with custom iterators. We showed that those iterators enabled C++ standard library algorithms to manipulate `MyArray` elements. We also introduced non-type template parameters for passing compile-time constants to templates and discussed default template arguments.

We demonstrated that variadic templates can receive any number of parameters, first using the standard library's `tuple` variadic class template, then with variadic function templates. We also showed how to apply binary operators to variadic parameter packs using fold expressions.

Many of the techniques we demonstrated are newer ways to perform various aspects of compile-time metaprogramming. We concluded with an introduction to several other metaprogramming techniques, including how to compute values at compile-time with value metafunctions, how to perform compile-time conditional compilation with `constexpr if` and how to modify types at compile-time with type metafunctions.

In the next chapter, we introduce modules—one of C++20's "big four" features (along with ranges, concepts and coroutines), which provide a new way to organize your code, precisely control which declarations you expose to client code and encapsulate implementation details.

16. C++20 Modules: Large-Scale Development

Objectives

In this chapter, you'll:

- Understand the motivation for modularity, especially for large software systems.
- See how modules improve encapsulation.
- `import` standard library headers as module header units.
- Define a `module`'s primary interface unit.
- `export` declarations from a `module` to make them available to other translation units.
- `import` modules to use their `export`ed declarations.
- Separate a module's interface from its implementation by placing the implementation in a `:private` module fragment or a module implementation unit.
- See what compilation errors occur when you attempt to use non-exported module items.
- Use module partitions to organize modules into logical components.
- Divide a module into "submodules" so client-code developers can choose the portion(s) of a library they wish to use.
- Understand visibility vs. reachability of declarations.
- See how modules can reduce translation unit sizes and compilation times.

Outline

16.1 Introduction
16.2 Compilation and Linking Before C++20
16.3 Advantages and Goals of Modules
16.4 Example: Transitioning to Modules—Header Units
16.5 Modules Can Reduce Translation Unit Sizes and Compilation Times
16.6 Example: Creating and Using a Module
16.6.1 `module` Declaration for a Module Interface Unit
16.6.2 Exporting a Declaration
16.6.3 Exporting a Group of Declarations
16.6.4 Exporting a `namespace`
16.6.5 Exporting a `namespace` Member
16.6.6 Importing a Module to Use Its Exported Declarations
16.6.7 Example: Attempting to Access Non-Exported Module Contents
16.7 Global Module Fragment
16.8 Separating Interface from Implementation
16.8.1 Example: Module Implementation Units
16.8.2 Example: Modularizing a Class
16.8.3 `:private` Module Fragment
16.9 Partitions
16.9.1 Example: Module Interface Partition Units
16.9.2 Module Implementation Partition Units
16.9.3 Example: "Submodules" vs. Partitions
16.10 Additional Modules Examples
16.10.1 Example: Importing the C++ Standard Library as Modules
16.10.2 Example: Cyclic Dependencies Are Not Allowed

16.10.3 Example: `import`s Are Not Transitive
16.10.4 Example: Visibility vs. Reachability
16.11 Migrating Code to Modules
16.12 Future of Modules and Modules Tooling
16.13 Wrap-Up
Appendix: Modules Videos Bibliography
Appendix: Modules Articles Bibliography
Appendix: Modules Glossary

16.1 Introduction

20 Mod **Modules**—one of C++20's "big four" features (along with ranges, concepts and coroutines)—provide a new way to organize your code, precisely control which declarations you expose to client code and encapsulate implementation details.[1] Each module is a uniquely named, reusable group of related declarations and definitions with a well-defined interface that client code can use.

---

1. Daveed Vandevoorde, "Modules in C++ (Revision 6)," January 11, 2012. https://wg21.link/n3347.

This chapter presents many complete, working code examples that introduce modules. Modules help developers be more productive, especially as they build, maintain and evolve **large software systems**.[2] Modules also aim to make such systems more scalable.[3] C++ creator Bjarne Stroustrup says, "Modules offer a historic opportunity to improve code hygiene and compile times for C++ (bringing C++ into the 21st century)."[4]

---

2. Gabriel Dos Reis, "Programming in the Large with C++ 20: Meeting C++ 2020 Keynote," December 11, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=j4du4LNsLiI.
3. Vassil Vassilev1, David Lange1, Malik Shahzad Muzaffar, Mircho Rodozov, Oksana Shadura and Alexander Penev, "C++ Modules in ROOT and Beyond," August 25, 2020. Accessed February 4, 2022. https://arxiv.org/pdf/2004.06507.pdf.
4. Bjarne Stroustrup, "Modules and Macros," February 11, 2018. Accessed February 4, 2022. https://wg21.link/p0955r0.

Even in small systems, modules can offer immediate benefits in every program. For example, you can replace C++ standard library `#include` preprocessor directives with `import` statements. This eliminates repeated processing of `#include`d content because modules-style `import`ed headers (Section 16.4), and modules in general, are **compiled once**, then reused where you import them in the program.

Brief History of Modules

C++ modules were challenging to implement, and the effort took many years. The first designs were proposed in the early 2000s, but the C++ committee was focused on what eventually became C++11. In 2014, modularization efforts continued with the proposal "A Module System for C++," based on Gabriel Dos Reis's and Bjarne Stroustrup's work at Texas A&M University.[5,6] The C++ committee formed a modules study group that led to the Modules TS (technical specification).[7] Most subsequent committee discussions focused on technical compromises and the specification details for C++20's modules capabilities.[8]

5. Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules,” June 12, 2020. Accessed February 6, 2022. https://www.stroustrup.com/hopl20-main-p5-p-bfc9cd4--final.pdf.
6. Gabriel Dos Reis, Mark Hall and Gor Nishanov, “A Module System for C++,” May 27, 2014. Accessed February 4, 2022. https://wg21.link/n4047.
7. Gabriel Dos Reis (ed.), “Working Draft, Extensions to C++ for Modules,” January 29, 2018. Accessed February 4, 2022. https://wg21.link/n4720.
8. Bjarne Stroustrup, “Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules,” June 12, 2020. Accessed February 6, 2022. https://www.stroustrup.com/hopl20-main-p5-p-bfc9cd4--final.pdf.

Compiler Support for Modules

At the time of this writing, our preferred compilers did not fully support modules. We tried each live-code example on each compiler. We present the required compilation commands for each example and indicate which compiler(s), if any, did not support a given feature.[9]

9. The compilation steps might change. If you encounter problems, email us at deitel@deitel.com. We’ll respond promptly and post updates on the book’s website: https://deitel.com/cpp20fp.

Docker Containers for `g++` 11.2 and `clang++` 13

To test our modules examples in the most current `g++` and `clang++` versions, we used the following free **Docker containers** from https://hub.docker.com:

- For `g++`, we used the official GNU GCC container’s latest version (11.2):

```
docker pull gcc:latest
```

- The LLVM/Clang team does not have an official Docker container, but many working containers are available on hub.docker.com. We used the most recent and widely downloaded one containing `clang++` version 13:[10]

10. The version of `clang++` in Xcode does not have as many of the C++20 features implemented as the version directly from the LLVM/Clang team. Also, at the time of this writing, using `"latest"` rather than `"13"` in the `docker pull` command gives you a Docker container with `clang++` 12, not 13.

```
docker pull silkeh/clang:13
```

If you’re not familiar with running Docker containers, see the book’s Before You Begin section and the Docker overview and getting started instructions at

```
https://docs.docker.com/get-started/overview/
```

C++ Compilation and Linking Before C++20

In Section 16.2, we’ll discuss the traditional C++ compilation and linking process and various problems with that model.

Advantages and Goals of Modules

In Section 16.3, we'll point out various modules benefits, some of which correct problems with and improve upon the pre-C++20 compilation process.

Future of Modules and Modules Tooling

Section 16.12 discusses and provides references for some C++23 modules features that are under development.

Videos and Articles Bibliographies; Modules Glossary

For your further study, we provide appendices at the end of this chapter containing lists of videos, articles, papers and documentation we referenced as we wrote this chapter. We also provide a modules glossary with key modules terms and definitions.

16.2 Compilation and Linking Before C++20

C++ has always had a **modular architecture for managing code** via a combination of **header files** and **source-code files**:

- Under the guidance of **preprocessor directives**, the **preprocessor** performs **text substitutions** and other text manipulations on each **source-code file**. A preprocessed source-code file is called a **translation unit**.[11]

11. "Translation Unit (Programming)." Wikipedia. Wikimedia Foundation. Accessed February 4, 2022. https://en.wikipedia.org/wiki/Translation_unit_(programming).

- The compiler converts each **translation unit** into an **object-code file**.
- The **linker** combines a program's **object-code files** with library object files, such as those of the **C++ standard library**, to create a program's **executable**.

This approach has been around since the 1970s. C++ inherited it from C.[12] If you're not familiar with the preprocessor, you might want to read online Appendix D, Preprocessor, before reading this chapter.

12. Gabriel Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer," October 5, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=tjSuKOz5HK4.

Problems with the Current Header-File/Source-Code-File Model

The preprocessor is simply a **text-substitution mechanism**—it does not understand C++. As motivation for C++20 modules, C++ creator Bjarne Stroustrup calls out three preprocessor problems:[13]

13. Stroustrup, "Thriving in a Crowded and Changing World."

- Err One header's contents can affect any subsequent `#include`d header, so **the order of `#include`s is important** and can cause subtle errors.
- Err A given C++ entity can have **different declarations in multiple translation units**. The compiler and linker do not always report such problems.
- Perf Perf **Reprocessing the same `#include`d content is slow**, particularly in large programs where a header can be included dozens or even hundreds of times. In each separate **translation unit**, the preprocessor will insert an `#include`d header's contents, possibly causing

the same code to be compiled repeatedly in many translation units. Eliminating this reprocessing by instead `import`ing headers as **header units** (Section 16.4) can significantly improve compilation times in large codebases.
Other preprocessor problems include:[14]

14. Bryce Adelstein Lelbach, "Modules Are Coming," May 1, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=yee9i2rUF3s.

- Definitions in headers can violate C++'s **One Definition Rule (ODR)**. The ODR says, "No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, template, default argument for a parameter (for a function in a given scope), or default template argument."[15]

15. "C++20 Standard: 6 Basics—6.3 One-Definition Rule." Accessed February 4, 2022. https://tim-song-cpp.github.io/cppwp/n4861/basic.def.odr.

- **Headers do not offer encapsulation**—everything in a header is available to the translation unit that `#include`s the header.
- **Accidental cyclic dependencies** between headers cause compilation errors and other problems.[16]

16. "Circular Dependency." Wikipedia. Wikimedia Foundation. Accessed February 4, 2022. https://en.wikipedia.org/wiki/Circular_dependency.

- Err Headers often define macros—`#define` preprocessor directives that create constants represented as symbols (such as `#define SIZE 10`), as well as function-like operations (such as `#define SQUARE(x) ((x) * (x))`).[17] Macros are handled by the preprocessor via text substitutions. The compiler cannot check their syntax and cannot report **multiple-definition errors** if two or more headers define the same macro name.

17. The all capital letters naming for preprocessor constants (like `SIZE`) and macros (like `SQUARE`) is a convention that some programmers prefer.

16.3 Advantages and Goals of Modules

Some benefits of using modules include:[18,19],20,21,22

18. Corentin Jabot, "What Do We Want from a Modularized Standard Library?," May 16, 2020. Accessed February 4, 2022. https://wg21.link/p2172r0.
19. Gabriel Dos Reis and Pavel Curtis, "Modules, Componentization, and Transition," October 5, 2015. Accessed February 4, 2022. https://wg21.link/p0141r0.
20. Daniela Engert, "Modules: The Beginner's Guide," May 2, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=Kqo-jIq4V3I.
21. Rainer Grimm, "C++20: The Advantages of Modules," May 10, 2020. Accessed February 4, 2022. https://www.modernescpp.com/index.php/cpp20-modules.
22. Dmitry Guzeev, "A Few Words on C++ Modules," January 8, 2018. Accessed February 4, 2022. https://medium.com/@dmitrygz/brief-article-on-c-modules-f58287a6c64.

- SE Better organization and componentization of large codebases.
- Smaller translation unit sizes.
- Perf Reduced compilation times.[23]

23. Rene Rivera, "Are Modules Fast? (Revision 1)," March 6 2019, Accessed February 4, 2022. https://wg21.link/p1441r1.

- Eliminating repetitive `#include` processing—**a compiled module does not have to be reprocessed for every source-code file that uses it.**
- Eliminating `#include` ordering issues.
- Eliminating many preprocessor directives that can introduce subtle errors.
- Eliminating **One Definition Rule (ODR)** violations.

Cons of Modules

Some disadvantages of using modules include:[24,25]

24. Steve Downey, "Writing a C++20 Module," July 5, 2021. Accessed February 4, 2022. https://www.youtube.com/watch?v=AO4piAqV9mg.

25. "C++ Modules Might Be Dead-on-Arrival," January 27, 2019. Accessed February 4, 2022. https://vector-of-bool.github.io/2019/01/27/modules-doa.html.

- Modules support is incomplete in most C++ compilers at the time of this writing.
- Existing codebases will need to be modified to fully benefit from modules.
- Modules do not solve C++ problems with respect to packaging and distributing software conveniently, as we can do with the package managers in several other popular languages.
- Compiled modules have compiler-specific aspects that are not portable across compilers and platforms, so you still need to distribute modules as source code.
- Modules uptake initially will be slow as organizations cautiously review the capabilities, decide how to structure new codebases, potentially modify existing ones and wait for information about the experiences of others.
- Few modules recommendations and guidelines currently exist. For example, the C++ Core Guidelines have not yet been updated for modules.

16.4 Example: Transitioning to Modules—Header Units

SE One goal of modules is to **eliminate the need for the preprocessor**. There are enormous numbers of preexisting libraries. Most libraries today are provided as:

- **header-only libraries**,
- a combination of **headers and source-code files**, or
- a combination of **headers and platform-specific object-code files**.

Mod It will take time for library developers to modularize their libraries. Some might never be modularized. As a transitional step from the preprocessor to modules, you can `import` (most) existing headers from the **C++ standard library**,[26] as shown in line 3 of Fig. 16.1. Doing so treats that header as a **header unit**.

26. "C++20 Standard: 10 Modules—10.3 Import Declaration." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.import.

```
1  // fig16_01.cpp
2  // Importing a standard library header as a header unit.
3  import <iostream>; // instead of #include <iostream>
4
5  int main() {
6     std::cout << "Welcome to C++20 Modules!\n";
7  }
```

```
Welcome to C++20 Modules!
```

Fig. 16.1 Importing a standard library header as a header unit.

How Header Units Differ from Header Files

Perf Rather than the preprocessor including the header's contents into a source-code file, the compiler processes the header as a **translation unit**, compiling it and producing information to treat the header as a module. In large-scale systems, this **improves compilation performance** because the header is compiled once, rather than having its contents inserted into every translation unit that includes the header.[27] Header units are similar to **precompiled headers** in some C++ environments.[28]

27. Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer."

28. Cameron DaCamara, "Practical C++20 Modules and the Future of Tooling Around C++ Modules," May 4, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=ow2zV0Udd9M.

Unlike `#include` directives, the order of `import`s is irrelevant. So, for example[29]

29. Stroustrup, "Thriving in a Crowded and Changing World."

```
import SomeModule;
import SomeOtherModule;
```

produces the same results as

```
import SomeOtherModule;
import SomeModule;
```

A **header unit**'s declarations are available to the **importing translation unit** because **header units implicitly "export" their contents**.[30] You'll see in Section 16.6 that the modules you define can specify which declarations they **export** (i.e., make available) for use in other translation units, giving you **precise control over each module's interface**. Unlike an `#include`, an `import` statement does not add source code to the `import`ing translation unit. Also, preprocessor directives in a translation unit that appear before you `import` a header unit do not affect the header unit's contents.[31]

30. "C++20 Standard: 10 Modules—10.3 Import Declaration." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.import.

31. "Understanding C++ Modules: Part 3: Linkage and Fragments," October 7, 2019. Accessed February 4, 2022. https://vector-of-bool.github.io/2019/10/07/modules-3.html.

Import All Headers as Header Units (If Possible)

SE Err **You should generally `import` all headers as header units.**[32] Unfortunately, not all headers can be imported. You'll typically use `#include` if importing a header as a header unit produces compilation errors. This could happen if a header depends on **`#defined` preprocessor macros**—referred to as **preprocessor state**.

32. Daniela Engert, "Modules: The Beginner's Guide," May 2, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=Kqo-jIq4V3I.

Compiling with Header Units in Microsoft Visual Studio

To compile any of this chapter's examples that use **header units**, ensure that your Visual Studio project is configured correctly:

1. Right-click the project name in **Solution Explorer** and select **Properties**.
2. In the **Property Pages** dialog, select **All Configurations** from the **Configurations** drop-down.

3. In the left column, under **Configuration Properties > C/C++ > Language**, set the **C++ Language Standard** option to **ISO C++20 Standard (/std:c++20)**.[33]

33. If the C++20 option does not work, try setting this to **Preview - Features from the Latest C++ Working Draft (/std:c++latest)**.

4. In the left column, under **Configuration Properties > C/C++ > All Options**, set the **Scan Sources for Module Dependencies** option to **Yes**.

5. Click **Apply**, then click **OK**.

Add `fig16_01.cpp` to your project's **Source Files** folder, then compile and run the project.

Compiling with Header Units in `g++`

In `g++`, you must first **compile each header** you'll `import` as a **header unit**.[34,35,36] The following command compiles the `<iostream>` **standard library header** as a header unit:

34. This might change once C++20 modules are fully implemented in `g++`.

35. "3.23 C++ Modules." Accessed February 4, 2022. https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

36. Nathan Sidwell, "C++ Modules: A Brief Tour," October 19, 2020. Accessed February 4, 2022. https://accu.org/journals/overload/28/159/sidwell/.

```
g++ -fmodules-ts -x c++-system-header iostream
```

- The `-fmodules-ts` **compiler flag** is currently required rather than `-std=c++20` to compile any code that uses C++20 modules.
- The `-x c++-system-header` **compiler flag** indicates that we are compiling a **C++ standard library header** as a **header unit**. You specify the header's name without the angle brackets.[37]

37. There are also `c++-header` and `c++-user-header` flags for precompiling other headers as header units. See https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

Next, compile the source-code file `fig16_01.cpp` using the following command:

```
g++ -fmodules-ts fig16_01.cpp -o fig16_01
```

This produces an executable named `fig16_01`, which you can execute with the command:

```
./fig16_0
```

Compiling with Header Units in clang++

Use the following command to compile this example in `clang++` version 13:[38]

38. This might change once C++20 modules are fully implemented in `clang++`. Also, header units work for standard library headers. User-defined headers require creating your own module maps. For more information, see https://clang.llvm.org/docs/Modules.html#module-maps.

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps fig16_01.cpp -o fig16_01
```

- The flag `-std=c++20` indicates that we're using C++20 language features.
- On some systems, `g++`'s C++ standard library is the default C++ library. The flag `-stdlib=libc++` ensures that `clang++`'s C++ standard library is used.

- The flags `-fimplicit-modules` and `-fimplicit-module-maps` enable `clang++` to generate and find the information it needs to treat headers as **header units**.

This produces an executable named `fig16_01`, which you can execute with the command

```
./fig16_01
```

16.5 Modules Can Reduce Translation Unit Sizes and Compilation Times

As we said, modules can reduce compilation times by eliminating repeated preprocessing of the same header files across many translation units in the same program. Consider the following simple program, which has fewer than 90 characters, including the vertical spacing and indentation:

```
#include <iostream>

int main() {
   std::cout << "Welcome to C++20 Modules!\n";
}
```

When you compile this program, the preprocessor inserts the contents of `<iostream>` into the translation unit. Each of our preferred compilers has a flag that enables you to see the result of preprocessing a source-code file:

- `-E` in `g++` and `clang++`
- `/P` in Visual C++

We used these flags to preprocess the preceding program. The **preprocessed translation unit sizes** on our system were:

- 1,023,010 bytes in g++,
- 1,883,270 bytes in clang++, and
- 1,497,116 bytes in Visual C++.

The preprocessed translation units are approximately 11,000 to 21,000 times the size of the original source file—a massive increase in the number of bytes per translation unit. Now imagine a large project with **thousands of translation units** that each `#include` **many headers**. Every `#include` must be preprocessed to form the translation units that the compiler then processes.

Perf **When using header units, each header is processed only once as a translation unit**. Also, importing a **header unit does not insert the header's contents into each translation unit**. Together, these significantly reduce compilation times.

References

For more information on translation unit sizes and compilation performance, see the following resources:

- **Gabriel Dos Reis and Cameron DaCamara, "Implementing C++ Modules: Lessons Learned, Lessons Abandoned,"** December 18, 2021. Accessed February 5, 2022. https://www.youtube.com/watch?v=9OWGgkuyFV8.

- **Bjarne Stroustrup**, "**Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules**," June 12, 2020. Accessed February 6, 2022. https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf.
- **Cameron DaCamara**, "**Practical C++20 Modules and the Future of Tooling Around C++ Modules**," May 4, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=ow2zV0Udd9M.
- **Rainer Grimm**, "**C++20: The Advantages of Modules**," May 10, 2020. Accessed February 4, 2022. https://www.modernescpp.com/index.php/cpp20-modules.
- **Rene Rivera**, "**Are Modules Fast? (Revision 1)**," March 6, 2019. Accessed February 4, 2022. https://wg21.link/p1441r1.

Compiler Profiling Tools

For tools you can use to profile compilation performance, see the following resources:

- Kevin Cadieux, Helena Gregg and Colin Robertson, "Get Started with C++ Build Insights," November 3, 2019. Accessed February 4, 2022. https://docs.micro-soft.com/en-us/cpp/build-insights/get-started-with-cpp-build-insights.
- "Clang 13 Documentation: Target-Independent Compilation Options `-ftime-report` and `-ftime-trace`," Accessed February 4, 2022. https://clang.llvm.org/docs/ClangCommandLineReference.html#target-independent-compilation-options.
- "Profiling the C++ Compilation Process." Accessed February 4, 2022. https://stackoverflow.com/questions/13559818/profiling-the-c-compilation-process.

16.6 Example: Creating and Using a Module

Mod Next, let's define our first module. A **module's interface** specifies the module members that the module makes available for use in other **translation units**. You do this by **exporting the member's declaration** using the `export` keyword in one of four ways:[39]

39. "C++ Standard: 10 Modules—10.2 Export Declaration." Accessed February 4, 2022. https://tim-song-cpp.github.io/cppwp/n4861/module.interface.

- `export` **its declaration directly**,
- `export` **a group of declarations enclosed in braces (`{` and `}`)**,
- `export` **a `namespace`**, which may contain many declarations, and
- `export` **a `namespace` member**, which also exports the `namespace`'s name,[40] but not the `namespace`'s other members.

40. So you qualify its exported members with "*name*`::`".

Recall that if an identifier's first occurrence is its definition, it also serves as the identifier's declaration. So each item in the preceding list may export declarations or definitions.

16.6.1 `module` *Declaration for a Module Interface Unit*

Mod Figure 16.2 defines our first **module unit**[41]—a **translation unit** that is part of a module. When a module is composed of **one translation unit**, the **module unit** is commonly referred to simply as a **module**. The module in Fig. 16.2 contains four functions (lines 8–10, 14–16, 21–23 and 28–30) to demonstrate **exporting declarations** for use in other translation units. In the following subsections, we'll discuss the `export` and `namespace` **"wrappers"** around the functions in lines 14–16, 21–23 and 28–30.

41. “C++ Standard: 10 Modules—10.1 Module Units and Purviews.” Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.unit.

```
1  // Fig. 16.2: welcome.ixx
2  // Primary module interface for a module named welcome.
3  export module welcome; // introduces the module name
4
5  import <string>; // class string is used in this module
6
7  // exporting a function
8  export std::string welcomeStandalone() {
9     return "welcomeStandalone function called";
10 }
11
12 // exporting all items in the braces that follow export
13 export {
14    std::string welcomeFromExportBlock() {
15       return "welcomeFromExportBlock function called";
16    }
17 }
18
19 // exporting a namespace exports all items in the namespace
20 export namespace TestNamespace1 {
21    std::string welcomeFromTestNamespace1() {
22       return "welcomeFromTestNamespace1 function called";
23    }
24 }
25
26 // exporting an item in a namespace exports the namespace name, too
27 namespace TestNamespace2 {
28    export std::string welcomeFromTestNamespace2() {
29       return "welcomeFromTestNamespace2 function called";
30    }
31 }
```

Fig. 16.2 Primary module interface for a module named `welcome`.

Module Declaration and Module Naming

Line 3’s **`module` declaration** names the module `welcome`. **Module names** are lowercase identifiers (by convention) separated by dots (`.`)[42]—such as `deitel.time` or `deitel.math`, which we’ll use in subsequent examples. The dots do not have special meaning—in fact, there’s a proposal to remove dot-separated naming from modules.[43] Both `deitel.time` and `deitel.math` begin with “`deitel.`” but these modules are not “submodules” of a larger module named `deitel`. All declarations from the **module declaration** to the end of the translation unit are part of the **module purview**, as are declarations from all other units that make up the module.[44] We show multi-file modules in subsequent sections.

42. Corentin Jabot, “Naming Guidelines for Modules,” June 16, 2019. Accessed February 4, 2022. https://wg21.link/P1634R0.
43. Michael Spencer, “P1873R1: remove.dots.in.module.names,” September 17, 2019. https://wg21.link/p1873r1.
44. “C++ Standard: 10 Modules—10.1 Module Units and Purviews.”

Mod Mod When you precede a **`module` declaration** with **`export`**, it introduces a **module interface unit**, which specifies the module members that the client code can access. Every module has one **primary module interface unit** containing an **`export module` declaration** that introduces the module's name. You'll see in Section 16.9 that the **primary module interface unit** may be composed of **module interface partition units**.

Module Interface File Extension

Microsoft Visual C++ uses the **`.ixx` filename extension** for **module interface units**. To add a module interface unit to your Visual C++ project:

1. Right-click your project's **Source Files** folder and select **Add > Module...**.
2. In the **Add New Item** dialog, specify a filename (we used the name `welcome.ixx`), specify where you want to save the file and click **Add**.
3. Replace the default code with the code in Fig. 16.2.

You are not required to use the `.ixx` filename extension. If you name a module interface unit's file with a different extension, right-click the file in your project, select **Properties** and ensure that the file's **Item Type** is set to **C/C++ compiler**.

Module Interface File Extensions for `g++` and `clang++`

The `g++` and `clang++` compilers do not require special filename extensions for module interface units. At the end of this section, we'll show how to enable `g++` and `clang++` to compile `.ixx` files so you can use the same filename extensions for all three compilers.

Other Filename Extensions

Some common filename extensions you'll encounter when using C++20 modules include:

- **`.ixx`**—Microsoft Visual C++ filename extension for the **primary module interface unit**.
- **`.ifc`**—Microsoft Visual C++ filename extension for the compiled version of the **primary module interface unit**.[45]

45. DaCamara, "Practical C++20 Modules and the Future of Tooling Around C++ Modules."

- **`.cpp`**—Filename extension for the C++ source code in a translation unit, including module units.
- **`.cppm`**—A recommended `clang++` filename extension for **module units**. Visual C++ also recognizes this extension.
- **`.pcm`**—When you compile a **primary module interface unit** in `clang++`, the resulting file uses this extension.

16.6.2 Exporting a Declaration

Mod You must **`export` a declaration** to make it available outside the module. Line 8 of Fig. 16.2 applies **`export`** to a function definition, which exports the **function's declaration** (that is, its **prototype**) as part of the **module's interface**. All `exported` declarations must appear after a **`module` declaration** in a **translation unit** and must appear at either **file scope** (known as **global namespace scope**) or in a **named namespace's scope** (Section 16.6.5). The declarations in `export` statements must not have **internal linkage**,[46] which includes

46. "C++ Standard: 10 Modules—10.2 Export Declaration."

- `static` variables and functions at **global namespace scope** in a **translation unit**,

- `const` or `constexpr` **global variables** in a **translation unit** and
- identifiers declared in "**unnamed namespaces**."[47]

47. "Namespaces—Unnamed Namespaces." Accessed February 4, 2022. https://en.cppreference.com/w/cpp/language/namespace#Unnamed_namespaces.

Also, if a module defines any preprocessor macros, they're for use only in that module and cannot be `export`ed.[48]

48. Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer."

Defining Templates, `constexpr` Functions and `inline` Functions in Modules

Similar to headers, when you **define a template, `constexpr` function or `inline` function in a module**, you must `export` its complete definition so the compiler can access it wherever the module is imported.

16.6.3 Exporting a Group of Declarations

SE Rather than applying **`export`** to individual declarations, you can **`export`** a group of declarations in braces (lines 13–17). **Every declaration in the braces is exported**. The braces do not define a new scope, so identifiers declared in such a block continue to exist beyond the block's closing brace.

16.6.4 Exporting a `namespace`

Err SE A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will collide with a variable of the same name in a different scope, possibly creating a naming conflict and resulting in errors. C++ solves this problem with **`namespaces`**. Each `namespace` defines a scope in which identifiers and variables are placed. As you know, the C++ standard library's features are defined in the **`std` namespace**, which helps ensure that these identifiers do not conflict with identifiers in your own programs.

Defining and Exporting `namespaces`

Lines 20–24 define the namespace `TestNamespace1`. A `namespace`'s body is delimited by braces (`{}`). A `namespace` may contain constants, data, classes and functions. Definitions of `namespaces` must be placed at **global namespace scope** or must be **nested** in other `namespaces`. Unlike classes, `namespace` members may be defined in separate but identically named `namespace` blocks. For example, each C++ standard library header has a `namespace` block like

```
namespace std {
   // standard library header's declarations
}
```

Mod indicating that the header's declarations are in `namespace std`. **When you place `export` before a given `namespace` block, all the members in that block are `exported`**, but not those in separate `namespace` blocks of the same namespace.

Accessing `namespace` Members

As always, to use a member of a `namespace`, you must qualify the member's name with the `namespace` name and the scope resolution operator (`::`), as in the expression

```
TestNamespace1::welcomeFromTestNamespace1()
```

Alternatively, you can provide a **`using` declaration** or **`using` directive** before the member is used in the translation unit. A `using` declaration like

```
using TestNamespace1::welcomeFromTestNamespace1;
```

brings one identifier (`welcomeFromTestNamespace1`) into the scope where the directive appears. A `using` directive like

```
using namespace TestNamespace1;
```

SE brings all the identifiers from the specified `namespace` into the scope where the directive appears. Members of the same `namespace` can access one another directly without using a **`namespace` qualifier**.

*16.6.5 Exporting a **`namespace`** Member*

Mod It's also possible to **`export`** specific `namespace` members rather than an entire `namespace`, as shown in lines 27–31. In this case, the `namespace`'s name is also exported. This does not implicitly `export` the namespace's other members.

16.6.6 Importing a Module to Use Its Exported Declarations

Mod SE To use a module's **`exported` declarations** in a given translation unit, you must provide an **`import` declaration** (Fig. 16.3, line 4) at **global namespace scope** containing the module's name. The module's `exported` declarations are available from the **`import`** declaration to the end of the **translation unit**. **Importing a module does not insert the module's code into a translation unit.** So, unlike headers, **modules do not need include guards** (Section 9.7.3). Lines 7–10 call the four functions we exported from the `welcome` module (Fig. 16.2). For the functions defined in namespaces, lines 9 and 10 precede each function name with its `namespace`'s name and the scope resolution operator **(`::`)**. The program's output shows that we were able to call each of module `welcome`'s `exported` functions.

```
1  // fig16_03.cpp
2  // Importing a module and using its exported items.
3  import <iostream>;
4  import welcome; // import the welcome module
5
6  int main() {
7     std::cout << welcomeStandalone() << '\n'
8        << welcomeFromExportBlock() << '\n'
9        << TestNamespace1::welcomeFromTestNamespace1() << '\n'
```

```
10          << TestNamespace2::welcomeFromTestNamespace2() << '\n';
11    }
welcomeStandalone function called
welcomeFromExportBlock function called
welcomeFromTestNamespace1 function called
welcomeFromTestNamespace2 function called
```

Fig. 16.3 Importing a module and using its exported items.

Compiling This Example in Visual C++

In Visual C++, ensure that `fig16_03.cpp` is in your project's **Source Files** folder. Then, run your project to compile the module and the `main` application.

Compiling This Example in `g++`

In `g++`, execute the following commands, which might change once C++20 modules are finalized in this compiler:

1. Compile the `<string>` and `<iostream>` headers as **header units** because they're used in our module and our `main` application, respectively:

```
g++ -fmodules-ts -x c++-system-header string
g++ -fmodules-ts -x c++-system-header iostream
```

2. Compile the **module interface unit**—this produces the file `welcome.o`:

```
g++ -fmodules-ts -c -x c++ welcome.ixx
```

3. Compile the `main` application and link it with `welcome.o`—this command produces the **executable file** `fig16_03`:

```
g++ -fmodules-ts fig16_03.cpp welcome.o -o fig16_03
```

In Step 2:

- the `-c` option says to compile `welcome.ixx`, but not link it, and
- the `-x c++` option indicates that `welcome.ixx` is a C++ file.

The `-x c++` is required because `.ixx` is not a standard `g++` filename extension. If we name `welcome.ixx` as `welcome.cpp`, then the `-x c++` option is not required.

Compiling This Example in `clang++`

In `clang++`, execute the following commands, which might change once C++20 modules are finalized in this compiler:

1. Compile the **module interface unit** into a **precompiled module (`.pcm`) file**, which is specific to the `clang++` compiler:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ welcome.ixx
   -Xclang -emit-module-interface -o welcome.pcm
```

2. Compile the `main` application and link it with `welcome.pcm`—this command produces the **executable file** `fig16_03`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_03.cpp welcome.pcm -o fig16_03
```

In Step 1:

- the `-c` option says to compile `welcome.ixx`, but not link it, and
- the `-x c++` option indicates that `welcome.ixx` is a C++ file.

The `-x c++` is required because `.ixx` is not a standard `clang++` filename extension. If we name the file `welcome.cpp`, then the `-x c++` option is not required. In Step 2, the option

```
-fprebuilt-module-path=.
```

indicates where `clang++` can locate **precompiled module (`.pcm`) files**. The dot (`.`) is the current folder but could be a relative or full path to another location on your system.

16.6.7 Example: Attempting to Access Non-Exported Module Contents

Mod Mod SE **Modules do not implicitly `export` declarations**—this is known as **strong encapsulation**. Thus, you have precise control over the declarations you `export` for use in other translation units. Figure 16.4 defines a **primary module interface unit** (line 3) named `deitel.math` containing the namespace `deitel::math`,[49] which is not exported. In the namespace, we define two functions—`square` (lines 7–9) is exported and `cube` (lines 12–14) is not. **Exporting the function `square` implicitly exports the enclosing namespace's name but does not export the namespace's other members**. Since `cube` is not exported, other translation units cannot call it (as you'll see in Fig. 16.5). This is a key difference from headers—everything declared in a header can be used wherever you `#include` it.[50]

17 49. The namespace `deitel::math` defines a `deitel` namespace with a nested namespace `math`. The notation used here was introduced in C++17. We discuss nested `namespace`s in online Chapter 20.

50. Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer."

```
1  // Fig. 16.4: deitel.math.ixx
2  // Primary module interface for a module named deitel.math.
3  export module deitel.math; // introduces the module name
4
5  namespace deitel::math {
6     // exported function square; namespace deitel::math implicitly exported
7     export int square(int x) {
8        return x * x;
9     }
10
11    // non-exported function cube is not implicitly exported
12    int cube(int x) {
13       return x * x * x;
14    }
15 };
```

Fig. 16.4 Primary module interface for a module named `deitel.math`.

```
1   // fig16_05.cpp
2   // Showing that a module's non-exported identifiers are inaccessible.
3   import <iostream>;
4   import deitel.math; // import the deitel.math module
5
6   int main() {
7      // can call square because it's exported from namespace deitel::math,
8      // which implicitly exports the namespace
9      std::cout << "square(3) = " << deitel::math::square(3) << '\n';
10
11     // cannot call cube because it's not exported
12     std::cout << "cube(3) = " << deitel::math::cube(3) << '\n';
13  }
```

```
Build started...
1>------ Build started: Project: modules_demo, Configuration: Debug Win32 ---
---
1>Scanning sources for module dependencies...
1>deitel.math.ixx
1>fig16_05.cpp
1>Compiling...

1>deitel.math.ixx
1>fig16_05.cpp

1>C:\Users\pauldeitel\Documents\examples\ch16\fig16_04-
05\fig16_05.cpp(12,47): error C2039: 'cube': is not a member of 'deitel::math'

1>C:\Users\pauldeitel\Documents\examples\examples\ch16\fig16_04-05\de
itel.math.ixx(5): message : see declaration of 'deitel::math'

1>C:\Users\pauldeitel\Documents\examples\examples\ch16\fig16_04-
05\fig16_05.cpp(12,51): error C3861: 'cube': identifier not found
```

Fig. 16.5 Showing that a module's non-exported identifiers are inaccessible.

SE It's good practice in a module to **put exported identifiers in namespaces** to help **avoid name collisions** when multiple **modules export the same identifier**. We found in our research that **namespace names** typically mimic their **module names**[51]—so for the `deitel.math` module, we specified the namespace `deitel::math` (line 5).

51. Daniela Engert, "Modules: The Beginner's Guide," May 2, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=Kqo-jIq4V3I.

Err In Fig. 16.5, line 4 imports the `deitel.math` module. Line 9 calls the module's exported `deitel::math::square` function, qualifying the function name with its enclosing `namespace`'s name. This compiles because `square` was exported by the `deitel.math` module. Line 12, however, results in compilation errors—the `deitel.math` module did not export `cube`. The compilation errors in Fig. 16.5 are from Visual C++. We highlighted key error messages in bold and added vertical spacing for readability.

`g++` Error Messages

Err To see the `g++` error messages, execute the following commands, each of which we explained in Section 16.6.6:

1. `g++ -fmodules-ts -x c++-system-header iostream`
2. `g++ -fmodules-ts -c -x c++ deitel.math.ixx`
3. `g++ -fmodules-ts fig16_05.cpp deitel.math.o`

We highlighted the key error message in bold:

```
fig16_05.cpp: In function 'int main()':
fig16_05.cpp:12:49: error: 'cube' is not a member of 'deitel::math'
   12 | std::cout << "cube(e) = " << deitel::math::cube(3) << '\n';
      |                                                 ^~~~
```

`clang++` Error Messages

Err To see the `clang++` error messages, execute the following commands, each of which we explained in Section 16.6.6:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
-fimplicit-module-maps -x c++ deitel.math.ixx
-Xclang -emit-module-interface -o deitel.math.pcm
```

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
-fimplicit-module-maps -fprebuilt-module-path=.
fig16_05.cpp deitel.math.pcm -o fig16_05
```

We highlighted the key error message in bold:

```
fig16_05.cpp:12:49: error: no member named 'cube' in namespace 'deitel::math'
   std::cout << "cube(e) = " << deitel::math::cube(3) << '\n';
                               ~~~~~~~~~~~~~~~^
1 error generated.
```

16.7 Global Module Fragment

As we mentioned, some headers cannot be compiled as **header units** because they require **preprocessor state**, such as macros defined in your translation unit or other headers. Such headers can be `#include`d for use in a **module unit** by placing them in the **global module fragment**[52]

52. "C++ Standard: 10 Modules—10.4 Global Module Fragment." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.global.frag.

```
module;
```

Mod which you place at the beginning of a module unit, before the `module` declaration. **The global module fragment may contain only preprocessor directives**. A module interface unit can export a declaration that was `#include`d into the global module fragment, so other implementation units that `import` the module can use that declaration.[53] **Global module fragments** from all **module units** are

placed into the **global module**, which also contains non-modularized code in **non-module translation units**, such as the one containing `main`.
53. Dos Reis, “Programming with C++ Modules: Guide for the Working Software Developer.”

16.8 Separating Interface from Implementation

In Chapters 9 and 10, we defined classes using **`.h` headers** and **`.cpp` source-code files** to **separate a class’s interface from its implementation**. Modules also support **separating interface from implementation**:

- SE You can **split your interface and implementation into separate files** (shown in Sections 16.8.1–16.8.2).
- SE You can **define your interface and implementation in the one source file** (discussed in Section 16.8.3).

16.8.1 Example: Module Implementation Units

SE Mod Sometimes it’s helpful to break a large module into smaller, more manageable pieces—for example, when a team of developers is working on different aspects of the same module. You can **split a module definition into multiple source files**. Let’s use a **primary module interface unit** for a **module’s interface** and a separate source-code file for the **module’s implementation details**.

Primary Module Interface Unit

The `deitel.math` module’s **primary module interface unit** (Fig. 16.6) exports the `deitel::math` namespace (lines 7–10) containing a function prototype for the function `average`, which calculates the average of a `vector<int>`’s elements.

```
1   // Fig. 16.6: deitel.math.ixx
2   // Primary module interface for a module named deitel.math.
3   export module deitel.math; // introduces the module name
4
5   import <vector>;
6
7   export namespace deitel::math {
8      // calculate the average of a vector<int>'s elements
9      double average(const std::vector<int>& values);
10  };
```

Fig. 16.6 Primary module interface for a module named `deitel.math`.

Module Implementation Unit

Mod SE Mod All files containing **`module` declarations without the `export` keyword** (line 3 of Fig. 16.7) are **module implementation units** (typically defined in `.cpp` files).[54] These can **split larger modules into multiple source files to make the code more manageable**. Line 3 indicates that this file is a module implementation unit for the `deitel.math` module. **A module implementation unit implicitly imports the interface for the specified module name.** The compiler combines the primary module interface unit and its corresponding module implementation unit(s) into a single **named module** that other translation units can `import`.[55] Lines 10–13 implement the `average` function in a namespace,

which must have the same name as the one containing `average`'s prototype in the primary module interface unit (Fig. 16.6).
54. "C++20 Standard: 10 Modules—10.1 Module Units and Purviews." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module#unit.
55. "C++20 Standard: 10 Modules." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.

```
1   // Fig. 16.7: deitel.math-impl.cpp
2   // Module implementation unit for the module deitel.math.
3   module deitel.math; // this file's contents belong to module deitel.math
4
5   import <numeric>;
6   import <vector>;
7
8   namespace deitel::math {
9      // average function's implementation
10     double average(const std::vector<int>& values) {
11        double total{std::accumulate(values.begin(), values.end(), 0.0)};
12        return total / values.size();
13     }
14  };
```

Fig. 16.7 Module implementation unit for the module `deitel.math`.

Using the Module

The `main` program in Fig. 16.8 imports module `deitel.math` (line 7) and calls its `average` function (line 17) to calculate the average of the `integers vector`'s elements.

```
1   // fig16_08.cpp
2   // Using the deitel.math module's average function.
3   import <algorithm>;
4   import <iostream>;
5   import <iterator>;
6   import <vector>;
7   import deitel.math; // import the deitel.math module
8
9   int main() {
10     std::ostream_iterator<int> output(std::cout, " ");
11     std::vector integers{1, 2, 3, 4};
12
13     std::cout << "vector integers: ";
14     std::copy(integers.begin(), integers.end(), output);
15
16     std::cout << "\naverage of integer's elements: "
17        << deitel::math::average(integers) << '\n';
18  }
```

```
vector integers: 1 2 3 4
average of integer's elements: 2.5
```

Fig. 16.8 Using the `deitel.math` module's `average` function.

Compiling This Example in Visual C++

Add the `deitel.math.ixx` file to your **Visual C++ project**, as you did in Section 16.6.1. Ensure that your project includes in its **Source Files** folder:

- `deitel.math.ixx`—the **primary module interface unit**,
- `deitel.math-impl.cpp`[56]—the **module implementation unit**, and

56. We named the module implementation unit with `"-impl"` in the filename to support the compilation steps of the `g++` compiler, ensuring that each resulting `.o` file will have a unique name.

- `fig16_08.cpp`—the main application.

Then, simply run your project to compile the module and the `main` application.

Compiling This Example in `g++`

In `g++`, use the commands introduced in Section 16.6.6 to compile the standard library headers `<algorithm>`, `<iostream>`, `<iterator>`, `<numeric>` and `<vector>` as header units. Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Then, compile the module implementation unit:

```
g++ -fmodules-ts -c deitel.math-impl.cpp
```

We now have the object files `deitel.math.o` and `deitel.math-impl.o` representing the module's interface and implementation. Finally, compile the `main` application and link it with `deitel.math.o` and `deitel.math-impl.o`:

```
g++ -fmodules-ts fig16_08.cpp deitel.math.o deitel.math-impl.o
   -o fig16_08
```

Compiling This Example in `clang++`

In `clang++`, compile the primary module interface unit into a precompiled module (`.pcm`) file:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules

   -fimplicit-module-maps -x c++ deitel.math.ixx

   -Xclang -emit-module-interface -o deitel.math.pcm
```

Next, compile the **module implementation unit** into an **object file**:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fmodule-file=deitel.math.pcm
```

```
   deitel.math-impl.cpp
```
The option `-fmodule-file=deitel.math.pcm` specifies the name of the module's primary module interface unit. Finally, compile the `main` application and link it with the files `deitel.math-impl.o` and `deitel.math.pcm`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=. fig16_08.cpp
   deitel.math-impl.o deitel.math.pcm -o fig16_08
```

16.8.2 Example: Modularizing a Class

Let's define a simplified version of Chapter 9's `Time` class with its interface in a **primary module interface unit** and its implementation in a **module implementation unit**. We'll name our module `deitel.time` and place the class in the `deitel::time` namespace.

`deitel.time` Primary Module Interface Unit

The `deitel.time` module's primary module interface unit (Fig. 16.9) defines and `export`s the namespace `deitel::time` (lines 7–19) containing the `Time` class definition (lines 8–18).

```
1  // Fig. 16.9: deitel.time.ixx
2  // Primary module interface for a simple Time class.
3  export module deitel.time; // declare the primary module interface
4
5  import <string>; // rather than #include <string>
6
7  export namespace deitel::time {
8     class Time {
9     public:
10       // default constructor because it can be called with no arguments
11       explicit Time(int hour = 0, int minute = 0, int second = 0);
12
13       std::string toString() const;
14    private:
15       int m_hour{0}; // 0 - 23 (24-hour clock format)
16       int m_minute{0}; // 0 - 59
17       int m_second{0}; // 0 - 59
18    };
19 }
```

Fig. 16.9 Primary module interface for a simple `Time` class.

`deitel.time` Module Implementation Unit

The **`module`** declaration in line 4 of Fig. 16.10 indicates that `deitel.time-impl.cpp` is a **`deitel.time` module implementation unit**. The rest of the file defines class `Time`'s member functions. Line 8

```
1  // Fig. 16.10: deitel.time-impl.cpp
2  // deitel.time module implementation unit containing the
3  // Time class member function definitions.
```

```
4  module deitel.time; // module implementation unit for deitel.time
5
6  import <stdexcept>;
7  import <string>;
8  using namespace deitel::time;
9
10 // Time constructor initializes each data member
11 Time::Time(int hour, int minute, int second) {
12    // validate hour, minute and second
13    if ((hour < 0 || hour >= 24) || (minute < 0 || minute >= 60) ||
14       (second < 0 || second >= 60)) {
15       throw std::invalid_argument{
16          "hour, minute or second was out of range"};
17    }
18
19    m_hour = hour;
20    m_minute = minute;
21    m_second = second;
22 }
23
24 // return a string representation of the Time
25 std::string Time::toString() const {
26    using namespace std::string_literals;
27
28    return "Hour: "s + std::to_string(m_hour) +
29       "\nMinute: "s + std::to_string(m_minute) +
30       "\nSecond: "s + std::to_string(m_second);
31 }
```

Fig. 16.10 `deitel.time` module implementation unit containing the `Time` class member function definitions.

```
using namespace deitel::time;
```

gives this module implementation unit access to namespace `deitel::time`'s contents. However, any translation unit that imports the `deitel.time` module does not see this `using` directive. So, other translation units still must access our module's exported names with `deitel::time` or via their own `using` statements.

Using Class `Time` from the `deitel.time` Module

The program in Fig. 16.11 imports the `deitel.time` module (line 7) and uses class `Time`. For convenience, line 8 indicates that this program uses the module's `deitel::time` namespace, but you also can fully qualify the mention of class `Time` (lines 11 and 17), as in

```
deitel::time::Time
```

```
// fig16_11.cpp
// Importing the deitel.time module and using its Time class.
import <iostream>;
import <stdexcept>;
import <string>;

import deitel.time;
using namespace deitel::time;

int main() {
   const Time t{12, 25, 42}; // hour, minute and second specified

   std::cout << "Time t:\n" << t.toString() << "\n\n";

   // attempt to initialize t2 with invalid values
   try {
      const Time t2{27, 74, 99}; // all bad values specified
   }
   catch (const std::invalid_argument& e) {
      std::cout << "t2 not created: " << e.what() << '\n';
   }
}
```

```
Time t:
Hour: 12
Minute: 25
Second: 42

t2 not created: hour, minute or second was out of range
```

Fig. 16.11 Importing the `deitel.time` module and using its `Time` class.

Compiling This Example in Visual C++

Add `deitel.time.ixx` to your Visual C++ project, as in Section 16.6.1. Next, ensure that your project includes in its **Source Files** folder:

- `deitel.time.ixx`—the primary module interface unit,
- `deitel.time-impl.cpp`—the module implementation unit, and
- `fig16_11.cpp`—the main application file.

Then, simply run your project to compile the module and the **`main` application**.

Compiling This Example in `g++`

In `g++`, use the commands introduced in Section 16.6.6 to compile the standard library headers `<iostream>`, `<string>` and `<stdexcept>` as header units.

Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.time.ixx
```

Then, compile the module implementation unit:

```
g++ -fmodules-ts -c deitel.time-impl.cpp
```

We now have files named `deitel.time.o` and `deitel.time-impl.o` representing the `deitel.time` module's interface and implementation.

Finally, compile the `main` application source file and link it with `deitel.time.o` and `deitel.time-impl.o`:[57]

57. At the time of this writing, this example is not linking correctly in `g++`.

```
g++ -fmodules-ts fig16_11.cpp deitel.time.o deitel.time-impl.o
   -o fig16_11
```

Compiling This Example in `clang++`

In `clang++`, compile the primary module interface unit into a precompiled module (`.pcm`) file:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ deitel.time.ixx
   -Xclang -emit-module-interface -o deitel.time.pcm
```

Next, compile the module implementation unit into an object file:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fmodule-file=deitel.time.pcm
   deitel.time-impl.cpp
```

The option `-fmodule-file=deitel.math.pcm` specifies the name of the module's primary module interface unit. Finally, compile the `main` application and link it with `deitel.math-impl.o` and `deitel.math.pcm`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_11.cpp deitel.time-impl.o deitel.time.pcm -o fig16_11
```

16.8.3 `:private` Module Fragment

Modules also support separating **interface** from **implementation** in one **translation unit** by using a **`:private` module fragment** in the **primary module interface unit**, as in:[58,59,60]

58. "C++20 Standard: 10 Modules—Private module fragment." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.private.frag.

59. Cameron DaCamara, "Standard C++20 Modules Support with MSVC in Visual Studio 2019 Version 16.8—Private Module Fragments," September 14, 2020, Accessed February 5, 2022. https://devblogs.microsoft.com/cppblog/standard-c20-modules-support-with-msvc-in-visual-studio-2019-version-16-8/#private-module-fragments.

60. Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer."

```
export module name; // introduces the module name

// code that defines the primary module interface

// private module fragment for defining implementation details
module :private;

// implementation details
```

SE With this approach to defining a module, **the primary module interface unit must be the module's only module unit**. Changes to the implementation details in the **`:private` module fragment** do not affect this module's interface, nor do they affect other translation units that **`import`** this module.[61]

61. "C++20 Standard: 10 Modules—Private Module Fragment." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.private.frag.

SE Perf In an e-mail communication with Cameron DaCamara from Microsoft's Visual C++ Team, he said, "The cases where the **`:private` module fragment** should be used are when you want to have all of your compiled code and interface code together in the same translation unit. The way I think of the **`:private` module fragment** is that it is essentially a **module implementation unit** after '`module :private;`', but I don't need to compile a separate `.cpp` file in order to implement details of the interface. The other major benefit is that having all the code in a single interface can help guide your toolset's optimization decisions (perhaps making better ones) without fancy linker-based technology."[62]

62. From a February 10, 2022 e-mail interaction between Paul Deitel and Cameron DaCamara, Senior Software Engineer at Microsoft and part of the Visual C++ team.

16.9 Partitions

Mod Perf You can divide a module's interface and/or its implementation into smaller pieces called **partitions**.[63] When working on large projects, this can help you organize a module's components into smaller, more manageable translation units. Breaking a larger module into smaller translation units also can reduce compilation times in large systems. Only translation units that have changed and translation units that depend on those changes would need to be recompiled.[64] Whether items are recompiled would be determined by the compiler's modules tooling.[65] **The compiler aggregates a module's partitions into a single named module** for import into other **translation units**.

63. At the time of this writing, `clang++` does not yet support partitions.
64. DaCamara, "Practical C++20 Modules and the Future of Tooling Around C++ Modules."
65. Modules tooling is under development at the time of this writing and will continue to evolve over the next several years.

16.9.1 Example: Module Interface Partition Units

In this example, we'll create a `deitel.math` module that exports four functions in its **primary module interface unit**—`square`, `cube`, `squareRoot` and `cubeRoot`. We'll divide these functions into two **module interface partition units** (`powers` and `roots`) to show partition syntax. Then we'll aggregate their exported declarations into a single **primary module interface partition**.

`deitel.math:powers` Module Interface Partition Unit

Mod Line 3 of Fig. 16.12 indicates that the translation unit `deitel.math-powers.ixx` is a **module interface partition unit**. The notation `deitel.math:powers` specifies that the **partition name** is `"powers"`, and the partition is part of the module `deitel.math`. This **module interface partition** exports the `deitel::math` namespace, containing the functions `square` and `cube`. **Module partitions are not visible outside their module, so they cannot be imported into translation units that are not part of the same module.**[66]

66. "Modules—Module Partitions." Accessed February 4, 2022. https://en.cppreference.com/w/cpp/language/modules.

```
1  // Fig. 16.12: deitel.math-powers.ixx
2  // Module interface partition unit deitel.math:powers.
3  export module deitel.math:powers;
4
5  export namespace deitel::math {
6     double square(double x) {return x * x;}
7     double cube(double x) {return x * x * x;}
8  }
```

Fig. 16.12 Module interface partition unit `deitel.math:powers`.

`deitel.math:roots` Module Interface Partition Unit

Line 3 of Fig. 16.13 indicates that the translation unit `deitel.math-roots.ixx` is a **module interface partition unit** with the **partition name** `"roots"`. The partition belongs to module `deitel.math`. This partition exports the `deitel::math` namespace, containing the functions `squareRoot` and `cubeRoot`. There are several **rules to keep in mind for partitions**:

- Mod **All module interface partitions with the same module name are part of the same module** (in this case, `deitel.math`). **They are not implicitly known to one another**, and they **do not implicitly import the module's interface.**[67]

67. Dos Reis, "Programming with C++ Modules: Guide for the Working Software Developer."

- Mod **Partitions may be imported only into other module units that belong to the same module.**
- Mod **One module interface partition unit can `import` another from the same module to use the other partition's features.**

```
1  // Fig. 16.13: deitel.math-roots.ixx
2  // Module interface partition unit deitel.math:roots.
3  export module deitel.math:roots;
4
5  import <cmath>;
6
7  export namespace deitel::math {
8     double squareRoot(double x) { return std::sqrt(x); }
9     double cubeRoot(double x) { return std::cbrt(x); }
10 }
```

Fig. 16.13 Module interface partition unit `deitel.math:roots`.

`deitel.math` Primary Module Interface Unit

Figure 16.14 defines the `deitel.math.ixx` primary module interface unit. **Every module must have one primary module interface unit with an `export module` declaration that does not include a partition name** (line 4). Lines 8 and 9 `import` and `export` the module interface partition units:

- Each **`import`** is followed by a colon (`:`) and the name of a **module interface partition unit** (in this case, `powers` or `roots`).
- Placing the **`export`** keyword before **`import`** indicates that each **module interface partition unit**'s exported members also should be part of the `deitel.math` module's **primary module interface**.

```
1  // Fig. 16.14: deitel.math.ixx
2  // Primary module interface unit deitel.math exports declarations from
3  // the module interface partitions :powers and :roots.
4  export module deitel.math; // declares the primary module interface unit
5
6  // import and re-export the declarations in the module
7  // interface partitions :powers and :roots
8  export import :powers;
9  export import :roots;
```

Fig. 16.14 Primary module interface unit that exports declarations from the module interface partitions `:powers` and `:roots`.

SE **The users of your module cannot see its partitions.**[68]

68. "C++ Standard: 10 Modules—10.1 Module Units and Purviews." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.unit.

You also can **`export import` primary module interface units**. Let's assume we have modules named `A` and `B`. If module `A`'s primary module interface unit contains

```
export import B;
```

then a translation unit that imports `A` also imports `B` and can use its exported declarations.

If you **`export import` a header unit**, its preprocessor macros are available for use only in the **importing translation unit—they are not re-exported**. So, to use a macro from a header unit in a specific translation unit, you must explicitly `import` the header.

Using the `deitel.math` Module

Figure 16.15 imports the `deitel.math` module (line 4) and lines 9–12 use its exported functions to demonstrate that this **primary module interface** contains all the functions exported by the **`powers`** and **`roots`** **module interface partitions**.

```
1  // fig16_15.cpp
2  // Using the deitel.math module's functions.
3  import <iostream>;
4  import deitel.math; // import the deitel.math module
5
6  using namespace deitel::math;
7
8  int main() {
9     std::cout << "square(6): " << square(6)
10       << "\ncube(5): " << cube(5)
11       << "\nsquareRoot(9): " << squareRoot(9)
12       << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
```

```
13   }
square(6): 36
cube(5): 125
squareRoot(9): 3
cubeRoot(1000): 10
```

Fig. 16.15 Using the `deitel.math` module's functions.

Compiling This Example in Visual C++

Add the files `deitel.math-powers.ixx deitel.math-roots.ixx` and `deitel.math.ixx` to your **Visual C++ project** using the steps from Section 16.6.1, then add the file `fig16_15.cpp` to the project's **Source Files** folder. Run your project to compile the module and the `main` **application**.

Compiling This Example in `g++`

When building a module with partitions, you must build the partitions before the primary module interface unit. In `g++`, use the commands introduced in Section 16.6.6 to compile the standard library headers `<cmath>` and `<iostream>` as header units. Next, compile each module interface partition unit:

```
g++ -fmodules-ts -c -x c++ deitel.math-powers.ixx
g++ -fmodules-ts -c -x c++ deitel.math-roots.ixx
```

Then, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Finally, compile the `main` application and link it with `deitel.math-powers.o`, `deitel.math-roots.o` and `deitel.math.o`:

```
g++ -fmodules-ts fig16_15.cpp deitel.math-powers.o

   deitel.math-roots.o deitel.math.o -o fig16_15
```

16.9.2 Module Implementation Partition Units

Mod Perf **You also can divide module implementations into module implementation partition units** to define a module's implementation details across **multiple source-code files.**[69] Again, this can help you organize a module's components into smaller, more manageable translation units and possibly reduce compilation times in large systems. In a **module implementation partition**, the **`module` declaration** must not contain the `export` keyword:

69. Richard Smith, "Merging Modules—Section 2.2 Module Partitions," February 22, 2019. Accessed February 4, 2022. https://wg21.link/p1103r3.

`module` *ModuleName*:*PartitionName*;

Module implementation partition units do not implicitly `import` the primary module interface.[70] At the time of this writing, none of our preferred compilers support module implementation partitions, so we do not show them here.

70. C++ Standard, "10 Modules—10.1 Module Units and Purviews."

16.9.3 Example: "Submodules" vs. Partitions

SE Some libraries, like the C++ standard library, are quite large. Programmers using such a library might want the flexibility to import only portions of it. A library vendor can divide a library into **logical "submodules,"** each with its own **primary module interface unit**. These can be imported independently. In addition, library vendors can provide a **primary module interface unit** that aggregates the "submodules" by importing and re-exporting their interfaces. Let's reimplement Section 16.9's `deitel.math` module using only **primary module interface units** to demonstrate the flexibility this provides to users.

`deitel.math.powers` Primary Module Interface Unit

First, let's rename `deitel.math-powers.ixx` as `deitel.math.powers.ixx`. We used the convention "`-`*name*" previously to indicate that the `powers` partition was part of module `deitel.math`. In this program, `deitel.math.powers` is a **primary module interface unit** (Fig. 16.16). Rather than declaring a **module interface partition**, as in Fig. 16.12

```
1   // Fig. 16.16: deitel.math.powers.ixx
2   // Primary module interface unit deitel.math.powers.
3   export module deitel.math.powers;
4
5   export namespace deitel::math {
6      double square(double x) {return x * x;}
7      double cube(double x) {return x * x * x;}
8   }
```

Fig. 16.16 Primary module interface unit `deitel.math.powers`.

```
export module deitel.math:powers;
```

line 3 of Fig. 16.16 declares a **primary module interface unit** with a dot-separated name:

```
export module deitel.math.powers;
```

We can now independently import `deitel.math.powers` and use its functions (Fig. 16.17).

```
1   // fig16_17.cpp
2   // Using the deitel.math.powers module's functions.
3   import <iostream>;
4   import deitel.math.powers; // import the deitel.math.powers module
5
6   using namespace deitel::math;
7
8   int main() {
9      std::cout << "square(6): " << square(6)
10        << "\ncube(5): " << cube(5) << '\n';
11  }
```

```
square(6): 36
cube(5): 125
```

Fig. 16.17 Using the `deitel.math.powers` module's functions.

Compiling This Example in Visual C++

Add the `deitel.math.powers.ixx` and `fig16_17.cpp` files to your Visual C++ project, as you did in Section 16.6.1. Run the project to compile the code and run the application.

Compiling This Example in `g++`

Use the commands introduced in Section 16.6.6 to compile the standard library header `<iostream>` as a header unit. Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.powers.ixx
```

Then, compile the `main` application and link it with `deitel.math.powers.o`:

```
g++ -fmodules-ts fig16_17.cpp deitel.math.powers.o -o fig16_17
```

Compiling This Example in `clang++`

Compile the primary module interface unit into a precompiled module (`.pcm`) file:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ deitel.math.powers.ixx
   -Xclang -emit-module-interface -o deitel.math.powers.pcm
```

Then, compile the `main` application and link it with `deitel.math.powers.o`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_17.cpp deitel.math.powers.pcm -o fig16_17
```

`deitel.math.roots` Primary Module Interface Unit

Next, let's rename the file `deitel.math-roots.ixx` as `deitel.math.roots.ixx` and make `deitel.math.roots` a **primary module interface unit** (Fig. 16.18). Rather than declaring a **module interface partition**, as in Fig. 16.13

```
1  // Fig. 16.18: deitel.math.roots.ixx
2  // Primary module interface unit deitel.math.roots.
3  export module deitel.math.roots;
4
5  import <cmath>;
6
7  export namespace deitel::math {
8     double squareRoot(double x) {return std::sqrt(x);}
9     double cubeRoot(double x) {return std::cbrt(x);}
10 }
```

Fig. 16.18 Primary module interface unit `deitel.math.roots`.

```
export module deitel.math:roots;
```

line 3 of Fig. 16.18 declares a **primary module interface unit** with a dot-separated name:

```
export module deitel.math.roots;
```

We can now independently import `deitel.math.roots` and use its functions (Fig. 16.19).

```
1   // fig16_19.cpp
2   // Using the deitel.math.roots module's functions.
3   import <iostream>;
4   import deitel.math.roots; // import the deitel.math.roots module
5
6   using namespace deitel::math;
7
8   int main() {
9      std::cout << "squareRoot(9): " << squareRoot(9)
10        << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
11  }
squareRoot(9): 3
cubeRoot(1000): 10
```

Fig. 16.19 Using the `deitel.math.roots` module's functions.

Compiling This Example in Visual C++

Add the `deitel.math.roots.ixx` and `fig16_19.cpp` files to your Visual C++ project, as you did in Section 16.6.1. Run the project to compile the code and run the application.

Compiling This Example in `g++`

Use the commands introduced in Section 16.6.6 to compile the standard library headers `<iostream>` and `<cmath>` as header units. Next, compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.powers.ixx
```

Then, compile the `main` application and link it with `deitel.math.powers.o`:

```
g++ -fmodules-ts fig16_19.cpp deitel.math.powers.o -o fig16_19
```

Compiling This Example in `clang++`

Compile the primary module interface unit into a precompiled module (`.pcm`) file:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -x c++ deitel.math.roots.ixx
   -Xclang -emit-module-interface -o deitel.math.roots.pcm
```

Then, compile the `main` application and link it with `deitel.math.roots.o`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   fig16_19.cpp deitel.math.roots.pcm -o fig16_19
```

`deitel.math` Primary Module Interface Unit

SE Figures 16.16 and 16.18 are now separate modules. Their names `deitel.math.powers` and `deitel.math.roots` imply a **logical relationship** between them, and both modules export the `deitel::math` namespace, but they do not define one module. For convenience, we can **aggregate these separate modules in a primary module interface unit that `export imports` both "submodules,"** as shown in lines 7 and 8 of Fig. 16.20. We can then use all the functions from both "submodules" by importing `deitel.math` (Fig. 16.21).

```
1   // Fig. 16.20: deitel.math.ixx
2   // Primary module interface unit deitel.math aggregates declarations
3   // from "submodules" deitel.math.powers and deitel.math.roots.
4   export module deitel.math; // primary module interface unit
5
6   // import and re-export deitel.math.powers and deitel.math.roots
7   export import deitel.math.powers;
8   export import deitel.math.roots;
```

Fig. 16.20 Primary module interface unit `deitel.math` aggregates declarations from "submodules" `deitel.math.powers` and `deitel.math.roots`.

```
1   // fig16_21.cpp
2   // Using the deitel.math module's functions.
3   import <iostream>;
4   import deitel.math; // import the deitel.math module
5
6   using namespace deitel::math;
7
8   int main() {
9      std::cout << "square(6): " << square(6)
10        << "\ncube(5): " << cube(5)
11        << "\nsquareRoot(9): " << squareRoot(9)
12        << "\ncubeRoot(1000): " << cubeRoot(1000) << '\n';
13  }
```

```
square(6): 36
cube(5): 125
squareRoot(9): 3
cubeRoot(1000): 10
```

Fig. 16.21 Using the `deitel.math` module's functions.

SE With these "submodules" developers now have the flexibility to

- import `deitel.math.powers` to use only `square` and `cube`,
- import `deitel.math.roots` to use only `squareRoot` and `cubeRoot`, or
- import the aggregated module `deitel.math` to use all four functions.

Compiling This Example in Visual C++

Add the Fig. 16.16, 16.18 and 16.20 `.ixx` files and `fig16_21.cpp` to your Visual C++ project, as you did in Section 16.6.1. Run the project to compile the code and run the application.

Compiling This Example in `g++`

For these steps, we assume you're executing commands in the same folder where you compiled `deitel.math.powers.ixx` and `deitel.math.roots.ixx`. Compile the primary module interface unit:

```
g++ -fmodules-ts -c -x c++ deitel.math.ixx
```

Then, compile the `main` application and link it with `deitel.math.powers.o`, `deitel.math.roots.o` and `deitel.math.o`:

```
g++ -fmodules-ts fig16_21.cpp deitel.math.powers.o
   deitel.math.roots.o deitel.math.o -o fig16_21
```

Compiling This Example in `clang++`

For these steps, we assume you're executing commands in the same folder where you compiled `deitel.math.powers.ixx` and `deitel.math.roots.ixx`. Compile the primary module interface unit into a precompiled module (`.pcm`) file:

```
clang++ -c -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=.
   -x c++ deitel.math.ixx -Xclang -emit-module-interface
   -o deitel.math.pcm
```

Then, compile the `main` application and link it with `deitel.math.powers.pcm`, `deitel.math.roots.pcm` and `deitel.math.pcm`:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps -fprebuilt-module-path=. fig16_21.cpp
   deitel.math.pcm deitel.math.roots.pcm deitel.math.powers.pcm
   -o fig16_21
```

16.10 Additional Modules Examples

The next several subsections demonstrate additional modules concepts:

- importing the modularized Microsoft and `clang++` standard libraries,
- some module restrictions and the compilation errors you'll receive if you violate those restrictions, and
- the difference between module members that other translation units can use by name vs. module members that other translation units can use indirectly.

Perf Mod The C++ standard does not currently require compilers to provide a **modularized standard library**. Microsoft provides one for Visual C++, which is split into several modules, and `clang++` has a single-module version of the standard library. At the time of this writing, `g++` does not have a modularized standard library.

You can `import` the following modules into your Visual C++ projects and "potentially speed up compilation times depending on the size of your project":[71]

71. Colin Robertson and Nick Schonning, "Overview of Modules in C++," December 13, 2019. Accessed February 4, 2022. https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160.

- `std.core`—This module contains most of the standard library, except for the following items.
- `std.filesystem`—Module containing the `<filesystem>` header's capabilities.
- `std.memory`—Module containing the `<memory>` header's capabilities.
- `std.regex`—Module containing the `<regex>` header's capabilities.
- `std.threading`—Module containing the capabilities of all the concurrency-related headers: `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` and `<thread>`.

For `clang++`, you can `import` the entire C++ standard library with

```
import std;
```

There is a proposal for two standard library modules named `std` and std.compat that might be included in C++23.[72]

72. Stephan T. Lavavej, Gabriel Dos Reis, Bjarne Stroustrup and Jonathan Wakely, "Standard Library Modules `std` and `std.compat`," October 13, 2021. Accessed February 4, 2022. https://wg21.link/p2465.

SE In Visual C++, you cannot combine code that **`#include`s standard library headers** with code that **`import`s Microsoft's standard library modules**. The compiler will not know which version of the standard libraries to choose when the final executable is linked.

Figure 16.22 imports the **`std.core` module** (line 3), then uses `cout` from the standard library's `<iostream>` capabilities to output a string.

Compiling the Program in Visual C++

Compiling a program that uses **Microsoft's modularized standard library** requires additional project settings, which may change once Microsoft's modules implementation is finalized. First, you must ensure that C++ modules support is installed:

1. In Visual Studio, select **Tools > Get Tools and Features...**.
2. In the **Individual Components** tab, search for "C++ Modules" and ensure that **C++ Modules for v### build tools** is checked—at the time of this writing, **###** is **143**. If not, check it, then click **Modify** to install it. You will need to close Visual Studio to complete the installation.

Next, you must configure several project settings:

1. In the **Solution Explorer**, right-click your project and select **Properties** to open the **Property Pages** dialog.
2. In the left column, select **Configuration Properties > C/C++ > Code Generation**.
3. In the right column, ensure that **Enable C++ Exceptions** is set to **Yes (/EHsc)**.
4. In the right column, ensure that **Runtime Library** is set to **Multi-threaded DLL (/MD)** if you are compiling in **Release** mode or **Multi-threaded Debug DLL (/MDd)** if you are compiling in **Debug** mode. You can choose settings for **Release** or **Debug** mode by changing the value in the **Configurations** drop-down at the top of the **Property Pages** dialog.
5. In the left column, select **Configuration Properties > C/C++ > Language**.

6. In the right column, ensure that **Enable Experimental C++ Standard Library Modules** is set to **Yes (/experimental:module)** and that **C++ Language Standard** is set to **ISO C++20 Standard (/std:c++20)**.
You can now compile and run Fig. 16.22.

```
1   // fig16_22.cpp
2   // Importing Microsoft's modularized standard library.
3   import std.core; // provides access to most of the C++ standard library
4
5   int main() {
6      std::cout << "Welcome to C++20 Modules!\n";
7   }
```

```
Welcome to C++20 Modules!
```

Fig. 16.22 Importing Microsoft's modularized standard library.

Modifying and Compiling the Program in `clang++`

To compile this program in `clang++`, change line 3 of Fig. 16.22 to

```
import std;
```

Then compile the program using the following command:

```
clang++ -std=c++20 -stdlib=libc++ -fimplicit-modules
   -fimplicit-module-maps fig16_22.cpp -o fig16_22
```

16.10.2 Example: Cyclic Dependencies Are Not Allowed

A module is not allowed to have a dependency on itself—that is, a module cannot import itself directly or indirectly.[73,74] Figures 16.23 and 16.24 define **primary module interface units** `moduleA` and `moduleB`—`moduleA` imports `moduleB`, and vice versa. Each module indirectly has a dependency on itself due to the `import` statements in line 5 of Fig. 16.23 and line 5 of Fig. 16.24—`moduleA` imports `moduleB`, which in turn imports `moduleA`.

73. "C++ Standard: 10 Modules—10.3 Import Declaration." Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.import.
74. "Understanding C++ Modules: Part 2: `export`, `import`, Visible, and Reachable," March 31, 2019. Accessed February 4, 2022. https://vector-of-bool.github.io/2019/03/31/modules-2.html.

```
1   // Fig. 16.23: moduleA.ixx
2   // Primary module interface unit that imports moduleB.
3   export module moduleA; // declares the primary module interface unit
4
5   export import moduleB; // import and re-export moduleB
```

Fig. 16.23 Primary module interface unit that imports `moduleB`.

```
1   // Fig. 16.24: moduleB.ixx
2   // Primary module interface unit that imports moduleA.
```

```
3   export module moduleB; // declares the primary module interface unit
4
5   export import moduleA; // import and re-export moduleA
```

Fig. 16.24 Primary module interface unit that imports `moduleA`.

Err Compiling Figs. 16.23 and 16.24 in Visual C++ results in the following error message:

```
error : Cannot build the following source files because there is a
cyclic dependency between them: ch16\fig16_23-24\moduleA.ixx depends
on ch16\fig16_23-24\moduleB.ixx depends on ch16\fig16_23-24\
moduleA.ixx.
```

You cannot compile this example in `g++` or `clang++` because each compiler requires a primary module interface unit to be compiled before you can import it. Since each module depends on the other, this is not possible.

16.10.3 Example: `import`s Are Not Transitive

Mod In Section 16.6.7, we mentioned that modules have **strong encapsulation** and **do not `export` declarations implicitly**. Thus, **`import` statements are not transitive**—if a **translation unit** imports `A` and `A` imports `B`, the **translation unit** that imported `A` does not automatically have access to `B`'s exported members.

Consider `moduleA` (Fig. 16.25) and `moduleB` (Fig. 16.26)—**`moduleB` imports but does not re-`export` `moduleA`** (line 6 of Fig. 16.26). As a result, **`moduleA`'s exported `cube` function is not part of `moduleB`'s interface**.

```
1   // Fig. 16.25: moduleA.ixx
2   // Primary module interface unit that exports function cube.
3   export module moduleA; // declares the primary module interface unit
4
5   export int cube(int x) { return x * x * x; }
```

Fig. 16.25 Primary module interface unit that exports function `cube`.

```
1   // Fig. 16.26: moduleB.ixx
2   // Primary module interface unit that imports, but does not export,
3  // moduleA and exports function square.
4   export module moduleB; // declares the primary module interface unit
5
6   import moduleA; // import but do not export moduleA
7
8   export int square(int x) { return x * x; }
```

Fig. 16.26 Primary module interface unit that imports, but does not export, `moduleA` and exports function `square`.

Err Figure 16.27 imports `moduleB` and attempts to use `moduleA`'s `cube` function (line 8), which generates errors because `cube`'s declaration is not imported. The key error messages produced by Visual C++, `g++` and `clang++`, respectively, are:

- `error C3861: 'cube': identifier not found`
- `error: 'cube' was not declared in this scope`
- `error: declaration of 'cube' must be imported from module 'moduleA' before it is required`

```
1  // fig16_27.cpp
2  // Showing that moduleB does not implicitly export moduleA's function.
3  import <iostream>;
4  import moduleB;
5
6  int main() {
7     std::cout << "square(6): " << square(6) // exported from moduleB
8        << "\ncube(5): " << cube(5) << '\n'; // not exported from moduleB
9  }
```

Fig. 16.27 Showing that `moduleB` does not implicitly export `moduleA`'s function.

16.10.4 Example: Visibility vs. Reachability

Mod Every example so far in which we used a name **`export`**ed from a module demonstrated the concept of **visibility**. **A declaration is visible in a translation unit if you can use its name**. As you've seen, any name **`export`**ed from a module can be used in translation units that **`import`** the module.

Mod Some declarations are **reachable** but not **visible**, meaning you **cannot explicitly mention the declaration's name in another translation unit, but the declaration is indirectly accessible.**[75,76,77] **Anything visible is reachable, but not vice versa**. The easiest way to understand this concept is with code. To demonstrate **reachability**, we modified Fig. 16.9's primary module interface unit `deitel.time.ixx`.[78] There are two key changes (Fig. 16.28):

75. "C++ Standard: 10 Modules—10.7 Reachability." Accessed February 4, 2022. https://timsongcpp.github.io/cppwp/n4861/module.reach.
76. "Understanding C++ Modules: Part 2: `export`, `import`, Visible, and Reachable."
77. Richard Smith, "Merging Modules—Section 2.2 Module Partitions," February 22, 2019. Accessed February 4, 2022. https://wg21.link/p1103r3.
78. At the time of this writing, this example does not link correctly in `g++` and does not compile in `clang++`.

- We no longer `export` namespace `deitel::time` (line 7), so **class `Time` is not `export`ed** and thus **not visible to translation units that `import deitel.time`**.
- We added an `export`ed function `getTime` (line 21) that **returns a `Time` object to its caller**—we'll use this function's return value to demonstrate **reachability**.

The module's implementation unit (Fig. 16.10) is identical for this example, so we do not show it here.

```
1  // Fig. 16.28: deitel.time.ixx
2  // Primary module interface unit for the deitel.time module.
3  export module deitel.time; // declare the primary module interface
4
5  import <string>; // rather than #include <string>
6
7  namespace deitel::time {
8     class Time { // not exported
9     public:
10       // default constructor because it can be called with no arguments
11       explicit Time(int hour = 0, int minute = 0, int second = 0);
```

```
12
13          std::string toString() const;
14       private:
15          int m_hour{0}; // 0 - 23 (24-hour clock format)
16          int m_minute{0}; // 0 - 59
17          int m_second{0}; // 0 - 59
18       };
19
20       // exported function returns a valid Time
21       export Time getTime() {return Time(6, 45, 0);}
22    }
```

Fig. 16.28 Primary module interface unit for the `deitel.time` module.

The program in Fig. 16.29 imports the `deitel.time` module (line 5). Line 10 calls the module's exported `getTime` function to get a `Time` object. Note that **we infer variable `t`'s type**. If you were to replace **`auto`** in line 10 with **`deitel::time::Time`**, you'd get an error like the following (from Visual C++):

```
'Time': is not a member of 'deitel::time'
```

```
1   // fig16_29.cpp
2   // Showing that type deitel::time::Time is reachable
3   // and its public members are visible.
4   import <iostream>;
5   import deitel.time;
6
7   int main() {
8      // initalize t with the object returned by getTime; cannot declare t
9      // as type Time because the type is not exported, and thus not visible
10     auto t{deitel::time::getTime()};
11
12     // Time's toString function is reachable, even though
13     // class Time was not exported by module deitel.time
14     std::cout << "Time t:\n" << t.toString() << "\n\n";
15  }
```

```
Time t:
Hour: 6
Minute: 45
Second: 0
```

Fig. 16.29 Showing that type `deitel::time::Time` is reachable and its `public` members are visible.

Mod This error occurs because **`Time` is not visible in this translation unit**. However, `Time`'s definition is **reachable** because **`getTime` returns a `Time` object**—the compiler knows this, so it can infer variable's `t`'s type. **When a class definition is reachable, the class's members become visible.** So, even though `deitel.time` does not `export` class `Time`, this translation unit can still call `Time` member function `toString` (line 14) to get `t`'s `string` representation. The compilation steps for this program are the same as those in Section 16.8.2, except that the `main` program's filename is now `fig16_29.cpp`.

16.11 Migrating Code to Modules

We've frequently referred to the C++ Core Guidelines for advice and recommendations on the proper ways to use various language elements. At the time of this writing, modules technology is still new, the popular compilers' modules implementations are not complete, and the C++ Core Guidelines have not yet been updated with modules recommendations. There also are not many articles and videos discussing developers' experiences with migrating existing software systems to modules. Some of our favorites are listed here. The Cameron DaCamara (Microsoft) and Steve Downey (Bloomberg) videos provide the most recent tips, guidelines and insights. The Daniela Engert and Nathan Sidwell videos each demonstrate modularizing existing code, and the Yuka Takahashi, Oksana Shadura and Vassil Vassilev paper discusses their experiences with modularizing portions of the large CERN ROOT C++ codebase:

- **Cameron DaCamara**, "**Moving a Project to C++ Named Modules**," August 10, 2021. Accessed February 4, 2022. https://devblogs.microsoft.com/cppblog/moving-a-project-to-cpp-named-modules/.
- **Steve Downey**, "**Writing a C++20 Module**," July 5, 2021. Accessed February 4, 2022. https://www.youtube.com/watch?v=AO4piAqV9mg.
- **Daniela Engert, "Modules: The Beginner's Guide,"** May 2, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=Kqo-jIq4V3I.
- **Yuka Takahashi**, **Oksana Shadura** and **Vassil Vassilev**, "**Migrating Large Code-bases to C++ Modules**," August 22, 2019. Accessed February 4, 2022. https://arxiv.org/abs/1906.05092.
- **Nathan Sidwell**, "**Converting to C++20 Modules**," October 4, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=KVsWIEw3TTw.

We will post additional resources as they become available at

```
https://deitel.com/c-plus-plus-20-for-programmers
```

16.12 Future of Modules and Modules Tooling

The C++ standard committee has begun its work on C++23, for which one of the key items will be a **modular standard library**.[79] C++20 modules are so new that the tooling to help you use modules is under development and will continue to evolve over several years. You've already seen some tooling. For example, if you used Visual C++ in this chapter's examples, you saw that Visual Studio enables you to add modules to your projects, and its build tools can compile and link your modularized applications.

79. "To Boldly Suggest an Overall Plan for C++23," November 25, 2019. https://wg21.link/p0592r4

Many popular programming languages have module systems or similar capabilities:

- Java has the Java Platform Module System (JPMS), for which we wrote a chapter in our Java books and published an article in Oracle's *Java Magazine*.[80]

80. Paul Deitel, "Understanding Java 9 Modules," *Oracle Java Magazine*, September/October 2017. https://www.oracle.com/a/ocom/docs/corporate/java-magazine-sept-oct-2017.pdf.

- Python has a well-developed module system, which we used extensively in our Python books.[81,82]

81. Paul Deitel and Harvey Deitel, *Python for Programmers*, 2019. Pearson Education, Inc.
82. Paul Deitel and Harvey Deitel, *Intro to Python for Computer Science and Data Science: Learning to Program with AI, Big Data and the Cloud*, 2020. Pearson Education, Inc.

- Microsoft's .NET platform languages like C# and Visual Basic can modularize code using assemblies.

Wikipedia lists several dozen languages with modules capabilities.[83]

83. "Modular Programming." Wikipedia. Wikimedia Foundation. Accessed February 4, 2022. https://en.wikipedia.org/wiki/Modular_programming.

Many languages provide tooling to help you work with their modules systems and modularize your code. The following tooling might eventually appear in the C++ ecosystem:

- **Module-aware build tools** that manage compiling software systems (Visual C++ already has this)
- **Tools to produce cross-platform module interfaces** so developers can distribute a module interface description and object code, rather than source code
- **Dependency-checking tools** to ensure that required modules are installed
- **Module discovery tools** to determine which modules and versions are installed
- **Tools that visualize module dependencies**, showing you the relationships among modules in software systems
- **Module packaging and distribution tools** to help developers install modules and their dependencies conveniently across platforms
- And more

References

A July 2021 paper from Daniel Ruoso of Bloomberg[84] discusses various problems with code reuse and build systems today and is meant to encourage discussions regarding the future of C++ modules tooling. That paper and the other resources highlighted here list in reverse chronological order various C++ standard and third-party vendor opportunities for module-aware tools that will improve the C++ development process:

84. Daniel Ruoso, "Requirements for Usage of C++ Modules at Bloomberg," July 12, 2021. Accessed February 4, 2022. https://wg21.link/P2409R0.

- **Daniel Ruoso**, "**Requirements for Usage of C++ Modules at Bloomberg**," July 12, 2021. Accessed February 4, 2022. https://isocpp.org/files/papers/P2409R0.pdf.
- **Nathan Sidwell**, "**P1184: A Module Mapper**," July 10, 2020. Accessed February 4, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1184r2.pdf.
- **Rob Irving**, **Jason Turner** and **Gabriel Dos Reis**, "**Modules Present and Future**," June 18, 2020. Accessed February 4, 2022. https://cppcast.com/modules-gaby-dos-reis/.
- **Cameron DaCamara**, "**Practical C++20 Modules and the Future of Tooling Around C++ Modules**," May 4, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=ow2zV0Udd9M.
- **Nathan Sidwell**, "**C++ Modules and Tooling**," October 4, 2018. Accessed February 4, 2022. https://www.youtube.com/watch?v=4yOZ8Zp_Zfk.
- **Gabriel Dos Reis**, "**Modules Are a Tooling Opportunity**," October 16, 2017. Accessed February 4, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0822r0.pdf.

16.13 Wrap-Up

In this chapter, we introduced modules—one of C++20's new "big four" features. You saw that modules help you organize your code, precisely control which declarations you expose to client code and encapsulate implementation details. We discussed the advantages of modules, including how they make developers more productive, make systems more scalable and can reduce translation unit sizes and improve build times. We also pointed out some disadvantages.

The chapter presented many complete, working modules code examples. You saw that even small systems can benefit from modules technology by transitioning from using preprocessor `#include` directives to `import`ing standard library headers as header units. We created custom modules. We implemented a primary module interface unit to specify a module's client-code interface, then `import`ed that module into an application to use its `exported` members. We employed `namespace`s to avoid naming conflicts with other modules' contents. You saw that non-`export`ed members are not accessible by name in `import`-ing translation units.

We separated interface from implementation—first in a primary module interface unit by placing the implementation code in the `:private` module fragment, then by using a module implementation unit. We divided a module into partitions to organize its components into smaller, more manageable translation units. We showed that "submodules" are more flexible than partitions because partitions cannot be `import`ed into translation units that are not part of the same module.

We demonstrated how easy it is to `import` with a single statement either Microsoft's or `clang++`'s modularized standard library. We showed that cyclic module dependencies are not allowed and that `import`s are not transitive. We discussed the difference between visible declarations and reachable declarations, mentioning that anything visible is reachable, but everything reachable is not necessarily visible.

We provided resources with tips for migrating legacy code to modules—a subject of great interest to organizations considering deploying modules technology. Finally, we discussed the future of C++20 modules and some types of tooling that might appear in the next several years. In the following appendices, for your further study we provide lists of the videos, articles, technical papers and documentation that we referenced as we wrote this chapter. We also include a glossary with key modules-related terms and definitions.

Modules technology is important. Once supported by the right tools, modules will provide C++ developers with significant opportunities to improve the design, implementation and evolution of libraries and large-scale software systems.

But modules are new and few organizations have experience using them. Some organizations will modularize new software—but what about the four decades' worth of non-modularized legacy C++ software? Some of it will eventually be modularized. Some will never be, possibly because the people who built and understand the systems have moved on.

At Deitel & Associates, we work with many widely used programming languages. Based on our experience studying, writing about and teaching the Java Platform Module System (JPMS), for example, we believe the uptake on C++20 modules will be gradual. Java introduced JPMS in 2017. Compiler vendor JetBrains' 2021 developer survey showed that 72% of Java developers are still working in some capacity with Java 8—the version of Java before JPMS was introduced.[85] Organizations using C++ will likely proceed with caution, too. Many will wait to learn

about other organizations' experiences with modularizing large legacy codebases and launching new modularized software-development projects.

85. "The State of Developer Ecosystem 2021," July 15, 2021. Accessed February 4, 2022. https://www.jetbrains.com/lp/devecosystem-2021/.

In the next chapter, we present C++ techniques that enable you to take advantage of your system's multi-core architecture—concurrency, parallelism and the parallel standard-library algorithms.

Appendix: Modules Videos Bibliography

Videos are listed in reverse chronological order.

- **Gabriel Dos Reis and Cameron DaCamara**, "**Implementing C++ Modules: Lessons Learned, Lessons Abandoned**," December 18, 2021. Accessed February 5, 2022. https://www.youtube.com/watch?v=9OWGgkuyFV8.
- **Steve Downey**, "**Writing a C++20 Module**," July 5, 2021. Accessed February 4, 2022. https://www.youtube.com/watch?v=AO4piAqV9mg.
- **Daniela Engert**, "**The Three Secret Spices of C++ Modules**," July 1, 2021. Accessed February 4, 2022. https://www.youtube.com/watch?v=l_83lyxWGtE.
- **Sy Brand**, "**C++ Modules: Year 2021**," May 6, 2021. Accessed February 4, 2022. https://www.youtube.com/watch?v=YcZntyWpqVQ.
- **Gabriel Dos Reis**, "**Programming in the Large with C++ 20: Meeting C++ 2020 Keynote**," December 11, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=j4du4LNsLiI.
- **Marc Gregoire**, "**C++20: An (Almost) Complete Overview**," September 26, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=FRkJCvHWdwQ.
- **Cameron DaCamara**, "**Practical C++20 Modules and the Future of Tooling Around C++ Modules**," May 4, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=ow2zV0Udd9M.
- **Timur Doumler**, "**How C++20 Changes the Way We Write Code**," October 10, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=ImLFlLjSveM.
- **Daniela Engert**, "**Modules: The Beginner's Guide**," May 2, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=Kqo-jIq4V3I.
- **Bryce Adelstein Lelbach**, "**Modules Are Coming**," May 1, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=yee9i2rUF3s.
- **Pure Virtual C++ 2020 Conference**, April 30, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=c1ThUFISDF4
- "**Demo: C++20 Modules**," March 30, 2020. Accessed February 4, 2022. https://www.youtube.com/watch?v=6SKIUeRaLZE.
- **Daniela Engert**, "**Dr Module and Sister #include,"** December 5, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=OCFOTle2G-A.
- **Boris Kolpackov**, "**Practical C++ Modules**," October 18, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=szHV6RdQdg8.
- **Michael Spencer**, "**Building Modules**," October 6, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=L0SHHkBenss.
- **Gabriel Dos Reis**, "**Programming with C++ Modules: Guide for the Working Software Developer**," October 5, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=tjSuKOz5HK4.
- **Nathan Sidwell**, "**Converting to C++20 Modules**," October 4, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=KVsWIEw3TTw.

- **Gabriel Dos Reis**, "**C++ Modules: What You Should Know**," September 13, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=MP6SJEBt6Ss
- **Richárd Szalay**, "**The Rough Road Towards Upgrading to C++ Modules**," June 16, 2019. Accessed February 4, 2022. https://www.youtube.com/watch?v=XJxQs8qgn-c.
- **Nathan Sidwell**, "**C++ Modules and Tooling**," October 4, 2018. Accessed February 4, 2022. https://www.youtube.com/watch?v=4yOZ8Zp_Zfk.

Appendix: Modules Articles Bibliography

Articles are listed in reverse chronological order.

- **Cameron DaCamara**, "**Moving a Project to C++ Named Modules**," August 10, 2021. Accessed February 4, 2022. https://devblogs.microsoft.com/cppblog/moving-a-project-to-cpp-named-modules/.
- **Cameron DaCamara**, "**Using C++ Modules in MSVC from the Command Line Part 1: Primary Module Interfaces**," July 21, 2021. Accessed February 4, 2022. https://devblogs.micro-soft.com/cppblog/using-cpp-modules-in-msvc-from-the-command-line-part-1/.
- **Daniel Ruoso**, "**Requirements for Usage of C++ Modules at Bloomberg**," July 12, 2021. Accessed February 4, 2022. https://wg21.link/P2409R0.
- **Andreas Fertig**, **Programming with C++20: Concepts, Coroutines, Ranges, and More**, 2021. https://andreasfertig.info/books/programming-with-cpp20/.
- **Nathan Sidwell**, "**C++ Modules: A Brief Tour**," October 19, 2020. Accessed February 4, 2022. https://accu.org/journals/overload/28/159/sidwell/.
- **Cameron DaCamara**, "**Standard C++20 Modules Support with MSVC in Visual Studio 2019 Version 16.8**," September 14, 2020. Accessed February 4, 2022. https://devblogs.microsoft.com/cppblog/standard-c20-modules-support-with-msvc-in-visual-studio-2019-version-16-8/.
- **Vassil Vassilev, David Lange, Malik Shahzad Muzaffar, Mircho Rodozov, Oksana Shadura and Alexander Penev**, "**C++ Modules in ROOT and Beyond**," August 25, 2020. Accessed February 4, 2022. https://arxiv.org/pdf/2004.06507.pdf.
- **Bjarne Stroustrup**, "**Thriving in a Crowded and Changing World: C++ 2006–2020—Section 9.3.1 Modules**," June 12, 2020. Accessed February 6, 2022. https://www.stroustrup.com/hopl20main-p5-p-bfc9cd4--final.pdf.
- **Rainer Grimm**, "**C++20: Further Open Questions to Modules**," June 8, 2020. Accessed February 4, 2022. https://www.modernescpp.com/index.php/c-20-open-questions-to-modules.
- **Rainer Grimm**, "**C++20: Structure Modules**," June 1, 2020. Accessed February 4, 2022. https://www.modernescpp.com/index.php/c-20-divide-modules.
- **Rainer Grimm**, "**C++20: Module Interface Unit and Module Implementation Unit**," May 25, 2020. Accessed February 4, 2022. https://www.modernescpp.com/index.php/c-20-module-interface-unit-and-module-implementation-unit.
- **Rainer Grimm**, "**C++20: A Simple Math Module**," May 17, 2020. Accessed February 4, 2022. https://www.modernescpp.com/index.php/cpp20-a-first-module.
- **Corentin Jabot**, "**What Do We Want from a Modularized Standard Library?**" May 16, 2020. Accessed February 4, 2022. https://wg21.link/p2172r0.
- **Rainer Grimm**, "**C++20: The Advantages of Modules**," May 10, 2020. Accessed February 4, 2022. https://www.modernescpp.com/index.php/cpp20-modules.
- **Cameron DaCamara**, "**C++ Modules Conformance Improvements with MSVC in Visual Studio 2019 16.5**," January 22, 2020. Accessed February 4, 2022. https://devblogs.microsoft.com/cppblog/c-modules-conformance-improvements-with-msvc-in-visual-studio-2019-16-5/.

- **Colin Robertson and Nick Schonning**, "**Overview of Modules in C++**," December 13, 2019. Accessed February 4, 2022. https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-160.
- **Arthur O'Dwyer**, "**Hello World with C++2a Modules**," November 7, 2019. Accessed February 4, 2022. https://quuxplusone.github.io/blog/2019/11/07/modular-hello-world/.
- "**Understanding C++ Modules: Part 3: Linkage and Fragments**," October 7, 2019. Accessed February 4, 2022. https://vector-of-bool.github.io/2019/10/07/modules-3.html.
- **Rainer Grimm**, "**More Details to Modules**," May 13, 2019. Accessed February 4, 2022. http://modernescpp.com/index.php/c-20-more-details-to-modules.
- **Rainer Grimm**, "**Modules**," May 6, 2019. Accessed February 4, 2022. http://modernescpp.com/index.php/c-20-modules.
- **Bryce Adelstein Lelbach** and **Ben Craig**, "**P1687R1: Summary of the Tooling Study Group's Modules Ecosystem Technical Report Telecons**," August 5, 2019. Accessed February 4, 2022. https://wg21.link/P1687R1.
- **Corentin Jabot**, "**Naming Guidelines for Modules**," June 16, 2019. Accessed February 4, 2022. https://wg21.link/P1634R0.
- "**Understanding C++ Modules: Part 2: `export`, `import`, Visible, and Reachable**," March 31, 2019. Accessed February 4, 2022. https://vector-of-bool.github.io/2019/03/31/modules-2.html.
- "**Understanding C++ Modules: Part 1: Hello Modules, and Module Units**," March 10, 2019. Accessed February 4, 2022. https://vector-of-bool.github.io/2019/03/10/modules-1.html.
- **Rene Rivera**, "**Are Modules Fast? (Revision 1)**," March 6, 2019, Accessed February 4, 2022. https://wg21.link/p1441r1.
- **Richard Smith**, "**Merging Modules**," February 22, 2019. Accessed February 4, 2022. https://wg21.link/p1103r3.
- "**C++ Modules Might Be Dead-on-Arrival**," January 27, 2019. Accessed February 4, 2022. https://vector-of-bool.github.io/2019/01/27/modules-doa.html.
- **Richard Smith and Gabriel Dos Reis**, "**Merging Modules**," June 22, 2018. Accessed February 4, 2022. https://wg21.link/p1103r0.
- **Gabriel Dos Reis and Richard Smith**, "**Modules for Standard C++**," May 7, 2018. Accessed February 4, 2022. https://wg21.link/p1087r0.
- **Richard Smith**, "**Another Take on Modules (Revision 1)**," March 6, 2018. Accessed February 4, 2022. https://wg21.link/p0947r1.
- **Bjarne Stroustrup**, "**Modules and Macros**," February 11, 2018. Accessed February 4, 2022. https://wg21.link/p0955r0.
- **Dmitry Guzeev**, "**A Few Words on C++ Modules**," January 8, 2018. Accessed February 4, 2022. https://medium.com/@dmitrygz/brief-article-on-c-modules-f58287a6c64.
- **Gabriel Dos Reis (ed.)**, "**Working Draft, Extensions to C++ for Modules**," January 29, 2018. Accessed February 4, 2022. https://wg21.link/n4720.
- **Gabriel Dos Reis and Pavel Curtis**, "**Modules, Componentization, and Transition**," October 5, 2015. Accessed February 4, 2022. https://wg21.link/p0141r0.
- **Gabriel Dos Reis, Mark Hall and Gor Nishanov**, "**A Module System for C++**," May 27, 2014. Accessed February 4, 2022. https://wg21.link/n4047.
- **Daveed Vandevoorde**, "**Modules in C++ (Revision 6)**," January 11, 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3347.pdf.

Documentation

- "**3.23 C++ Modules.**" Accessed February 4, 2022. https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Modules.html.

- “**C++20 Standard: 10 Modules**.” Accessed February 4, 2022. https://timsong-cpp.github.io/cppwp/n4861/module.
- “**Modules—Module Partitions**.” Accessed February 4, 2022. https://en.cppreference.com/w/cpp/language/modules.

Appendix: Modules Glossary

- `export` **a declaration**—Make a declaration available to translation units that `import` the corresponding module.
- `export` **a definition**—Make a definition (such as a template) available to **translation units** that `import the corresponding module`.
- `export` **followed by braces**—A module exports all the declarations or definitions in the block. The block does not define a new scope.
- `export module` **declaration**—Indicates that a module unit is the primary module interface unit and introduces the module’s name.
- **global module**—An unnamed module that contains all identifiers defined in non-module translation units or in global module fragments.
- **global module fragment**—In a module unit, this fragment may contain only preprocessor directives and must appear before the `module` declaration. All declarations in this fragment are part of the global module and can be used throughout the remainder of the module unit.
- **header unit**—A header that is `import`ed rather than `#include`d.
- **IFC (`.ifc`) format**—A Microsoft Visual C++ file format for storing the information the compiler generates for a module.
- `import` **a header file**—Enables existing headers to be processed as header units, which can reduce compilation times in large projects.
- `import` **a module**—Make a module’s `export`ed declarations available in a translation unit.
- `import` **declaration**—A statement used to `import` a module into a translation unit.
- **interface dependency**—If you import a module into an implementation unit, the implementation unit has an dependency on that module’s interface.
- `module` **declaration**—Every module unit has a `module` declaration specifying the module’s name and possibly a partition name.
- **module implementation unit**—A module unit in which the `module` declaration does not begin with the `export` keyword.
- **module interface unit**—A module unit in which the `module` declaration begins with the `export` keyword.
- **module linkage**—Names in a module that are exported from it are known only in that module.
- **module name**—The name specified in a module’s `module` declaration. All module units in a given module must have the same module name.
- **module partition**—A module unit in which the module declaration specifies the module name followed by a colon and a partition name. Module partition names in the same named module must be unique. If a module partition is a module interface partition, it must be exported by the module’s primary interface unit.
- **module purview**—The set of identifiers within a module unit from the `module` declaration to the end of the translation unit.
- **module unit**—An implementation unit containing a `module` declaration.
- **named module**—All module units with the same module name.
- **named module purview**—The purviews of all the module units in the named module.

- **namespace**—Defines a scope in which identifiers and variables are placed to help prevent naming conflicts with identifiers in your own programs and libraries.
- **partition**—A kind of module unit that defines a portion of a module's interface or implementation. Partitions are not visible to translation units that import the module.
- **precompiled module interface (`.pcm`)**—A `clang++` file that contains information about a module's interface. Used when compiling other translation units that depend on a given module.
- **primary module interface unit**—Determines the set of declarations exported by a module for use in other translation units.
- `:private` **module fragment**—A section in a primary module interface unit that enables you to define a module's implementation in the same file as its interface without exposing the implementation to other translation units.
- **reachable declaration**—A declaration is reachable if you can use it in your code without referencing it directly. For example, if you `import` a module into a translation unit and one of the module's `export`ed functions returns an object of a non-exported type, that type is reachable in importing translation units.
- **translation unit**—A preprocessed source-code file that is ready to be compiled.
- **visible declaration**—A declaration you can use by name in your code. For example, if you `import` a module into a translation unit, the module's exported declarations are visible in that translation unit.

17. Parallel Algorithms and Concurrency: A High-Level View

Objectives

In this chapter, you'll:

- Understand concurrency, parallelism and multithreading.
- Use high-level concurrency features such as C++17 parallel algorithms and C++20 latches and barriers.
- Understand the thread life cycle.
- Use the `<chrono>` header's timing features to profile sequential and parallel algorithm performance on multi-core systems.
- Implement correct producer–consumer relationships.
- Synchronize access to shared mutable data by multiple threads using `std::mutex`, `std::lock_guard`, `std::condition_variable` and `std::unique_lock`.
- Use `std::async` and `std::future` to execute long calculations asynchronously and get their results.
- Use C++20 concurrency features, including latches, barriers, semaphores and enhanced atomics.
- Learn about possible future concurrency features.

Outline

17.1 Introduction
17.2 Standard Library Parallel Algorithms (C++17)
17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms
17.2.2 When to Use Parallel Algorithms
17.2.3 Execution Policies
17.2.4 Example: Profiling Parallel and Vectorized Operations
17.2.5 Additional Parallel Algorithm Notes
17.3 Multithreaded Programming
17.3.1 Thread States and the Thread Life Cycle
17.3.2 Deadlock and Indefinite Postponement
17.4 Launching Tasks with `std::jthread`
17.4.1 Defining a Task to Perform in a Thread
17.4.2 Executing a Task in a `jthread`
17.4.3 How `jthread` Fixes `thread`
17.5 Producer–Consumer Relationship: A First Attempt
17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data
17.6.1 Class `SynchronizedBuffer`: Mutexes, Locks and Condition Variables
17.6.2 Testing `SynchronizedBuffer`
17.7 Producer–Consumer: Minimizing Waits with a Circular Buffer
17.8 Readers and Writers
17.9 Cooperatively Canceling `jthread`s
17.10 Launching Tasks with `std::async`
17.11 Thread-Safe, One-Time Initialization
17.12 A Brief Introduction to Atomics
17.13 Coordinating Threads with C++20 Latches and Barriers
17.13.1 C++20 `std::latch`

17.13.2 C++20 `std::barrier`
17.14 C++20 Semaphores
17.15 C++23: A Look to the Future of C++ Concurrency
17.15.1 Parallel Ranges Algorithms
17.15.2 Concurrent Containers
17.15.3 Other Concurrency-Related Proposals
17.16 Wrap-Up

17.1 Introduction

Perf It would be nice if we could focus our attention on performing only one task at a time and doing it well. That can be difficult to do in a complex world where so much is going on at once. This chapter presents C++'s features for building applications that create and manage multiple tasks. This can significantly improve program performance and responsiveness.

Sequential, Concurrent and Parallel Operation of Multiple Tasks

When we say that two tasks operate **sequentially**, we mean they operate one after the other "in sequence." When we say that two tasks are operating **concurrently**, we mean that they're both **making progress**, possibly in small increments and not necessarily simultaneously.

Let's clarify that. Until the early 2000s, most computers had only a single processor. Operating systems on such computers execute tasks concurrently by rapidly switching between them, doing a portion of each task before moving on to the next so that all tasks keep progressing. For example, it's common for personal computers to concurrently perform cloud backups, compile a program, send a file to a printer, send and receive email messages and tweets, stream video and audio, download files, and more.

When we say that two tasks operate **in parallel**, we mean they're **truly executing simultaneously**. In this sense, parallelism is a subset of concurrency. The human body performs a great variety of operations in parallel. Respiration, blood circulation, digestion, thinking and walking, for example, can occur in parallel, as can all the senses—sight, hearing, touch, smell and taste. It's believed that this parallelism is possible because the human brain is thought to contain billions of "processors." Today's multi-core computers have multiple processors that can perform tasks in parallel.

C++ Concurrency

C++ makes concurrency available to you through the language and standard libraries. Threads of execution are what happen concurrently in a program. According to the C++ standard, "a **thread of execution** (also known as a **thread**) is a single flow of control within a program, including the initial invocation of a specific top-level function, and recursively including every function invocation subsequently executed by the thread."[1]

---

1. C++ Standard, "Multi-Threaded Executions and Data Races." Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/intro.multithread#def:thread_of_execution).

Programs can have **multiple threads of execution**, each with its own function-call stack and program counter, allowing it to execute concurrently with other threads. All the threads in a given program can share application-wide resources, such as memory and files. This capability is called **multithreading**.

Perf A problem with **single-threaded applications** that can lead to poor responsiveness is that lengthy activities must complete before others can begin. In a **multithreaded application**, threads can be distributed across multiple cores (if available), enabling tasks to truly execute in parallel, and the application can operate more efficiently.

Perf Multithreading also can enhance performance on single-processor systems. When one thread cannot proceed (because, for example, it's waiting for the result of an I/O operation), another can use the processor.

A Concurrent Programming Use Case: Video Streaming

We'll discuss various concurrent programming applications. For example, when streaming video over the Internet, the user may not want to wait until an entire lengthy video finishes downloading before starting playback. Multiple threads can be used to solve this problem. One downloads the video a "chunk" at a time (we'll refer to this thread as a **producer**), and another plays it (we'll refer to this thread as a **consumer**). These activities can proceed concurrently. The threads are synchronized to avoid choppy playback. **Their actions are coordinated** so that the player thread doesn't begin until there's a sufficient portion of the video in memory to keep the player proceeding smoothly. Producer and consumer threads **share data**. We'll show how to synchronize threads to ensure correct execution. In the case of video streaming, synchronizing threads ensures smooth viewing, even though only a portion of the video might be in memory at once.

Thread Safety

When threads share mutable (modifiable) data, you must ensure that they do not corrupt it, which is known as making the code thread-safe. Thread safety approaches include:[2]

2. "Thread Safety." Accessed February 7, 2022. https://en.wikipedia.org/wiki/Thread_safety.

- **Immutable (constant) data**—A constant object is not modifiable, so any number of threads can access a constant object at once.
- **Mutual exclusion**—You can coordinate access to shared mutable data, allowing only one thread at a time to access the data.
- **Atomic types**—You can use low-level types that automatically ensure their operations are atomic (i.e., not interruptible), so only one thread at a time can access and possibly modify the data.
- 11 **Thread-local storage**—In code executed by multiple threads, declaring a `static` or global variable with the storage class `thread_local` (C++11) indicates that each thread should have its own copy of that variable. **Threads do not share variables declared `thread_local`, so these variables do not present thread-safety issues.**[3,4]

3. If you were to explicitly share pointers or references to `thread_local` variables, they could present thread-safety issues.
4. Paul E. McKenney and J. F. Bastien, "Use Cases for Thread-Local Storage," November 20, 2014. Accessed February 7, 2022. https://wg21.link/n4324.

Concurrent Programming Is Complex

Writing multithreaded programs can be tricky. Although the human mind can perform functions concurrently, people find it difficult to jump between parallel trains of thought. Try the following experiment to see why multithreaded programs can be challenging to write and understand. Open three books on dramatically different topics to page 1 and try reading the books concurrently. Read a

few words from the first, then a few from the second, then a few from the third, then loop back and read the next few words from the first, and so on. After this experiment, you'll appreciate some of multithreading's challenges—switching between the books, reading briefly, remembering your place in each book, moving the book you're reading closer to bring the text and images into focus and pushing aside the books you're not reading. And, amid all this chaos, trying to comprehend the content of the books!

11 C++11 and C++14: Providing Low-Level Concurrency Features

14 17 20 23 14 Before C++11, C++ multithreading libraries were non-standard, platform-specific extensions. C++11 introduced **standardized multithreading**. For C++11 and C++14, the C++ Standards Committee defined mostly low-level primitives. These capabilities were then used to build higher-level C++17 and C++20 features. They're also being used to implement the higher-level features coming in C++23 and later. C++11 was the first C++ version to include standard library features for implementing multithreaded applications, including low-level thread synchronization primitives called **mutexes** and **locks**.[5] C++14 added **shared mutexes** and **shared locks**.[6]

5. "C++11." Accessed February 7, 2022. https://en.wikipedia.org/wiki/C%2B%2B11.
6. "C++14." Accessed February 7, 2022. https://en.wikipedia.org/wiki/C%2B%2B14.

17 C++17 and C++20: Providing Convenient Higher-Level Concurrency Features

20 23 The higher-level concurrency features introduced in C++17 and C++20 are meant to simplify concurrent programming, as are features proposed for C++23, which we'll briefly introduce in Section 17.15. Higher-level concurrency features are essential because:

- They simplify concurrent programming.
- They help you find concurrency bugs in environments where it's typically impossible to reconstruct the exact circumstances in which each bug appears.
- They help you avoid common errors.
- They make your programs easier to maintain.

17 20 C++17 added 69 parallel standard library algorithms (mentioned in Sections 14.8 and 14.9).[7] C++20 added higher-level thread-synchronization capabilities (latches, barriers and semaphores—not available in `clang++` at the time of this writing), additional parallel algorithms and coroutines (discussed in Chapter 18, C++20 Coroutines).[8] The rest of this chapter introduces C++'s features for implementing multithreaded applications.

7. "C++17." Accessed February 7, 2022. https://en.wikipedia.org/wiki/C%2B%2B17.
8. "C++20." Accessed February 7, 2022. https://en.wikipedia.org/wiki/C%2B%2B20.

17 17.2 Standard Library Parallel Algorithms (C++17)

Perf Computer processing power continues to increase, but **Moore's law**[9] has essentially expired, so hardware vendors now rely on multi-core processors for better performance. In this chapter, **we emphasize high-level approaches to building concurrent applications**. We begin with C++17's parallel standard library algorithm overloads. These algorithms benefit from concurrent execution by taking advantage of multi-core architectures and high-performance "vector mathematics."[10, 11] **Vector operations** perform the same task on many data items simultaneously using the **SIMD (single instruction, multiple data) instructions** provided by many CPUs and GPUs.[12, 13]

9. "Moore's Law." Wikipedia, Wikimedia Foundation. Accessed February 10, 2022. https://en.wikipedia.org/wiki/Moore%27s_law.
10. Billy O'Neal (Visual C++ Team blog), "Using C++17 Parallel Algorithms for Better Performance," September 11, 2018. Accessed February 7, 2022. https://devblogs.microsoft.com/cppblog/using-c17-parallel-algorithms-for-better-performance/.
11. Dietmar Kuhl, "C++17 Parallel Algorithms," October 28, 2017. Accessed February 7, 2022. https://www.youtube.com/watch?v=Ve8cHE9LNfk.
12. C++ Standard, "General Utilities Library—Execution Policies—Unsequenced Execution Policy." Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/execpol.unseq.
13. "SIMD." Accessed February 7, 2022. https://en.wikipedia.org/wiki/SIMD.

17.2.1 Example: Profiling Sequential and Parallel Sorting Algorithms

Perf 20 23 Section 6.12 used the **`std::sort` algorithm** to sort a `std::array` in a **single thread of execution**. Let's compare this algorithm with its **parallel overload** to determine whether there's a performance improvement. Figure 17.1 sorts 100,000,000 randomly generated `int`s stored in `vector`s. We use **timing features from the `<chrono>` header** to demonstrate the performance improvement of parallel sorting vs. sequential sorting on a multi-core system. We compiled this program in Visual C++ and ran it on a Windows 10 64-bit computer using an 8-core Intel processor. Your results likely will differ based on your hardware, operating system and compiler, and the workload on your system when you run the example. Note that this example uses the `sort` algorithms that require **random-access iterators** rather than a **random-access range**. C++20's **`std::ranges` algorithms** are not yet parallelized, though they might be in C++23.[14]
14. Barry Revzin, Conor Hoekstra and Tim Song, "A Plan for C++23 Ranges," October 14, 2020. Accessed February 7, 2022. https://wg21.link/p2214r0.

```
1  // fig17_01.cpp
2  // Profiling sequential and parallel sorting with the std::sort algorithm.
3  #include <algorithm>
4  #include <chrono> // for timing operations
5  #include <execution> // for execution policies
6  #include <iostream>
7  #include <iterator>
8  #include <random>
9  #include <vector>
10
11 int main() {
12    // set up random-number generation
13    std::random_device rd;
14    std::default_random_engine engine{rd()};
15    std::uniform_int_distribution ints{};
16
17    std::cout << "Creating a vector v1 to hold 100,000,000 ints\n";
18    std::vector<int> v1(100'000'000); // 100,000,000 element vector
19
20    std::cout << "Filling vector v1 with random ints\n";
21    std::generate(v1.begin(), v1.end(), [&](){return ints(engine);});
22
23    // copy v1 to create identical data sets for each sort demonstration
24    std::cout << "Copying v1 to vector v2 to create identical data sets\n";
```

```
25      std::vector v2{v1};
26
27      // <chrono> library features we'll use for timing
28      using std::chrono::steady_clock;
29      using std::chrono::duration_cast;
30      using std::chrono::milliseconds;
31
32      // sequentially sort v1
33      std::cout << "\nSorting 100,000,000 ints sequentially\n";
34      auto start1{steady_clock::now()}; // get current time
35      std::sort(v1.begin(), v1.end()); // sequential sort
36      auto end1{steady_clock::now()}; // get current time
37
38      // calculate and display time in milliseconds
39      auto time1{duration_cast<milliseconds>(end1 - start1)};
40      std::cout << "Time: " << (time1.count() / 1000.0) << " seconds\n";
41
42      // parallel sort v2
43      std::cout << "\nSorting the same 100,000,000 ints in parallel\n";
44      auto start2{steady_clock::now()}; // get current time
45      std::sort(std::execution::par, v2.begin(), v2.end()); // parallel sort
46      auto end2{steady_clock::now()}; // get current time
47
48      // calculate and display time in milliseconds
49      auto time2{duration_cast<milliseconds>(end2 - start2)};
50      std::cout << "Time: " << (time2.count() / 1000.0) << " seconds\n";
51  }
```

```
Creating a vector v1 to hold 100,000,000 ints
Filling vector v1 with random ints
Copying v1 to vector v2 to create identical data sets

Sorting 100,000,000 ints sequentially
Time: 8.296 seconds

Sorting the same 100,000,000 ints in parallel
Time: 1.227 seconds
```

Fig. 17.1 Profiling sequential and parallel sorting with the `std::sort` algorithm.

Setting Up Random-Number Generation

Lines 13–15 set up the random-number capabilities we'll use in this example. We did not specify the range of random numbers, so by default, the `uniform_int_distribution` will generate integers in the range 0 to `std::numeric_limits<int>::max()`—the maximum `int` value on the system.

Creating the Arrays

Line 18 creates a `vector` to hold 100,000,000 `int`s, and line 21 uses **`std::generate`** to fill the `vector` with random `int` values. Line 25 copies the `vector`, so we can compare sequential and parallel sorting on `vector`s with identical contents.

Timing Operations with `std::chrono`

To time the sequential and parallel sorts, we'll use features from the C++ standard library's date and time utilities found in the **`<chrono>` header**. Lines 28–30 indicate that we're using namespace **`std::chrono`'s** **`steady_clock`**, **`duration_cast`** and **`milliseconds`**:

- We'll use a `steady_clock` object to get the time before a sorting operation starts and after it completes. An object of this type is **recommended for timing operations.**[15] If you need the time of day, you can use a `system_clock` object, which returns the time since January 1, 1970, at midnight UTC.[16]

15. "`std::chrono::steady_clock`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/chrono/steady_clock.
16. "`std::chrono::system_clock`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/chrono/system_clock.

- The `duration_cast` function template converts a duration into another measurement. For example, we'll calculate the duration between two times, then call `duration_cast` to convert the result to milliseconds.
- We'll use the `std::chrono::duration` type `milliseconds` and a `duration_cast` to determine the total execution time of each `sort` call.

Sequential Sorting

Lines 33–40 test sequential sorting and time the results. Lines 34 and 36 get the current time before and after the `sort` call (line 35). Line 39 calculates the difference between the `end1` and `start1` times, then converts the result to milliseconds, which it returns as a `std::chrono::duration` object. Line 40 calls the `duration`'s `count` member function to get the sort duration in milliseconds, then divides that by `1000.0` to display the result in seconds.

Parallel Sorting with Execution Policy `std::execution::par`

Lines 43–50 test parallel sorting and time the results. Lines 44 and 46 get the current time before and after the call to **`sort`'s parallel overload** (line 45). Each parallel algorithm overload requires as its first parameter an **execution policy** indicating whether to parallelize a task and, if so, how to do it. Line 45 calls `sort` with the **`std::execution::par` execution policy** (from header `<execution>`), which indicates that the algorithm should try to execute portions of its work simultaneously on multiple cores. Section 17.2.3 overviews the four standard execution policies.

Perf Line 49 calculates the difference between the `end2` and `start2` times, then gets the sort duration in milliseconds. Line 50 divides that by `1000.0` and displays the result in seconds. This program's output shows that **parallel execution was 6.76 times faster than sequential execution** on our system. You can see a **clear performance advantage** when parallel sorting a large volume of data on multiple cores.

Caution: Parallel Is Not Always Faster

Perf **You cannot simply assume that using parallel algorithms will improve performance. Sometimes parallel algorithms actually perform worse than the corresponding sequential algorithms.**[17] This is especially true when processing small numbers of elements and when using non-random-access iterators. In these cases, the overhead of parallelization may outweigh the benefits of parallel performance. Microsoft's C++ standard library implementation defaults some parallel algorithms to sequential because **benchmarking showed that the parallel versions ran slower for the kinds of hardware Visual C++ targets.**[18] For these reasons, Microsoft's parallel versions of algorithms `copy`, `copy_n`, `fill`, `fill_n`, `move`, `reverse`, `reverse_copy`, `rotate`, `rotate_copy` and `swap_ranges` default to sequential execution. Microsoft's Billy O'Neal recommended considering parallel algorithms for tasks that process at least 2,000 items and require more than $O(n)$ time (e.g., sorting).[19]

17. Lucian Radu Teodorescu, "A Case Against Blind Use of C++ Parallel Algorithms," February 4–7, 2021. Accessed February 7, 2022. https://accu.org/journals/overload/29/161/teodorescu/.

18. O'Neal, "Using C++17 Parallel Algorithms for Better Performance."

19. O'Neal, "Using C++17 Parallel Algorithms for Better Performance."

17.2.2 When to Use Parallel Algorithms

To show that **parallel execution might increase sort times for small datasets**, we ran the program of Fig. 17.1 with `vector`s of random integers from 100 to 100,000,000 elements in multiples of 10 and measured the execution times in nanoseconds. We used Visual C++ on two computers:

- our everyday four-core Windows 10, 64-bit system and
- a more powerful eight-core Windows 10, 64-bit system (on which we often run processor-intensive training of machine-learning and deep-learning artificial intelligence models).

The following table shows the results from both systems. On each, parallel execution started to (barely) outperform sequential execution at 10,000 random integers. The improvements became more significant as the number of elements increased.

Number of elements	4-core sequential execution (in ns)	4-core parallel execution (in ns)	8-core sequential execution (in ns)	8-core parallel execution (in ns)
100	3,200	81,900	2,800	63,400
1,000	42,500	161,900	33,300	136,400
10,000	880,400	711,400	433,900	431,600
100,000	10,205,300	6,308,200	5,888,300	1,289,500
1,000,000	98,959,700	27,816,100	75,358,800	12,486,400
10,000,000	1,065,163,900	415,386,000	814,166,200	126,300,900
100,000,000	12,361,988,600	3,444,056,800	8,473,599,400	1,230,407,100

17.2.3 Execution Policies

Figure 17.1 demonstrated the `std::execution::par` **execution policy**. There are four standard execution policies:[20]

20. "`std::execution::seq`, `std::execution::par`, `std::execution::par_unseq`, `std::execution::unseq`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag.

- 17 `std::execution::seq` (C++17) indicates that an algorithm must execute in a single thread. This is an object of class `std::execution::sequenced_policy`.
- 17 `std::execution::par` (C++17) indicates that an algorithm can be **parallelized**. This is an object of class `std::execution::parallel_policy`.
- 17 `std::execution::par_unseq` (C++17) indicates that an algorithm can be **parallelized** and **vectorized**. This is an object of class `std::execution::parallel_unsequenced_policy`. Again, **vector hardware operations perform the same task on many data items simultaneously**.
- 20 `std::execution::unseq` (C++20) indicates that an algorithm can be **vectorized**. This is an object of class `std::execution::unsequenced_policy`.

Compilers also can provide **custom execution policies.**[21]

21. C++ Standard, "20.18.1 Execution Policies." Accessed February 7, 2022. https://timsongcpp.github.io/cppwp/n4861/execpol#general.

Perf C++ is used on a wide range of devices and operating systems, some of which do not support parallelism. **These execution policies are just suggestions—compilers can ignore them or handle them differently**. For example, if a compiler targets hardware that does not support vectorization, the compiler might default the `par_unseq` policy to `par`. In fact, Visual C++ "implements the parallel and parallel unsequenced policies the same way, so you should not expect better performance when using `par_unseq`."[22] Test your code and compare its sequential, parallelized and vectorized algorithm performance to determine whether it makes sense to use the parallel algorithm overloads.

22. O'Neal, "Using C++17 Parallel Algorithms for Better Performance."

17.2.4 Example: Profiling Parallel and Vectorized Operations

The program of Fig. 17.2 uses `std::transform` to demonstrate parallel execution with `std::execution::par` vs. vectorized execution with `std::execution::unseq`:

- Lines 37–38 create `vector`s of 100 million and one billion elements, respectively, and lines 41–44 fill each with random `int` values.
- Lines 49–58 call the abbreviated function template `timeTransform` (lines 13–28) to calculate the duration of the `std::transform` operations on each `vector` using each execution policy. Lines 61–67 display the timing results.
- Function `timeTransform`'s first argument is the execution policy, which we pass to `std::transform` (line 21). Lines 21–22 calculate the square root of every element in `timeTransform`'s `vector` argument, writing the results into another `vector`.
- Lines 20 and 23 time the `std::transform` call, then lines 26–27 calculate and return the duration in seconds.

We compiled and ran this example using `g++` 11.2 on a MacBook Pro with an Intel processor. **On our system, using the `std::execution::unseq` execution policy required approximately half the execution time of `std::execution::par`.** As in Fig. 17.1, your results likely will differ based on your hardware, operating system and compiler, and the workload on your system when you run the example.

```
1  // fig17_02.cpp
2  // Performing transforms with execution policies par and unseq.
3  #include <algorithm>
4  #include <chrono> // for timing operations
5  #include <cmath>
6  #include <execution> // for execution policies
7  #include <fmt/format.h>
8  #include <iostream>
9  #include <random>
10 #include <vector>
11
12 // time each std::transform call and return its duration in seconds
13 double timeTransform(auto policy, const std::vector<int>& v) {
14    // <chrono> library features we'll use for timing
15    using std::chrono::steady_clock;
```

```
16      using std::chrono::duration_cast;
17      using std::chrono::milliseconds;
18
19      std::vector<double> result(v.size());
20      auto start{steady_clock::now()}; // get current time
21      std::transform(policy, v.begin(), v.end(),
22         result.begin(), [](auto x) {return std::sqrt(x);});
23      auto end{steady_clock::now()}; // get current time
24
25      // calculate and return time in seconds
26      auto time{duration_cast<milliseconds>(end - start)};
27      return time.count() / 1000.0;
28   }
29
30   int main() {
31      // set up random-number generation
32      std::random_device rd;
33      std::default_random_engine engine{rd()};
34      std::uniform_int_distribution ints{0, 1000};
35
36      std::cout << "Creating vectors\n";
37      std::vector<int> v1(100'000'000);
38      std::vector<int> v2(1'000'000'000);
39
40      std::cout << "Filling vectors with random ints\n";
41      std::generate(std::execution::par, v1.begin(), v1.end(),
42         [&]() {return ints(engine);});
43      std::generate(std::execution::par, v2.begin(), v2.end(),
44         [&]() {return ints(engine);});
45
46      std::cout << "\nCalculating square roots:\n";
47
48      // time the transforms on 100,000,000 elements
49      std::cout << fmt::format("{} elements with par\n", v1.size());
50      double parTime1{timeTransform(std::execution::par, v1)};
51      std::cout << fmt::format("{} elements with unseq\n", v1.size());
52      double unseqTime1{timeTransform(std::execution::unseq, v1)};
53
54      // time the transforms on 1,000,000,000 elements
55      std::cout << fmt::format("{} elements with par\n", v2.size());
56      double parTime2{timeTransform(std::execution::par, v2)};
57      std::cout << fmt::format("{} elements with unseq\n", v2.size());
58      double unseqTime2{timeTransform(std::execution::unseq, v2)};
59
60      // display table of timing results
61      std::cout << "\nExecution times (in seconds):\n\n"
62         << fmt::format("{:>13}{:>17}{:>21}\n", "# of elements",
63               "par (parallel)", "unseq (vectorized)")
64         << fmt::format("{:>13}{:>17.3f}{:>21.3f}\n",
65               v1.size(), parTime1, unseqTime1)
66         << fmt::format("{:>13}{:>17.3f}{:>21.3f}\n",
67               v2.size(), parTime2, unseqTime2);
68   }
Creating vectors
Filling vectors with random ints
```

```
Calculating square roots:
100000000 elements with par
100000000 elements with unseq
1000000000 elements with par
1000000000 elements with unseq

Execution times (in seconds):

# of elements   par (parallel)  unseq (vectorized)
    100000000            2.401               1.215
   1000000000           22.969              11.787
```

Fig. 17.2 Performing transforms with execution policies `par` and `unseq`.

17.2.5 Additional Parallel Algorithm Notes

17 In addition to adding parallel overloads for 69 existing algorithms, C++17 added seven new parallel algorithms:

- `for_each_n`[23] applies a function to the first *n* elements of a range.

23. "`std::for_each_n`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/algorithm/for_each_n.

- `exclusive_scan`[24] and `inclusive_scan`[25] are parallelized versions of algorithm `partial_sum` (introduced in Section 14.4.13). The parallel algorithms differ in that `exclusive_scan` does not include the *n*th input element when calculating the *n*th sum, but `inclusive_scan` does. Like `partial_sum`, these parallel algorithms can be customized to use binary operations other than addition.

24. "`std::exclusive_scan`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/algorithm/exclusive_scan.

25. "`std::inclusive_scan`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/algorithm/inclusive_scan.

- `transform_exclusive_scan`[26] and `transform_inclusive_scan`[27] are the same as `exclusive_scan` and `inclusive_scan`, respectively, but apply a transformation function to each element before calculating the sums (or other binary operation).

26. "`std::transform_exclusive_scan`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/algorithm/transform_exclusive_scan.

27. "`std::transform_inclusive_scan`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/algorithm/transform_inclusive_scan.

- `reduce` produces a single value from a range of values (e.g., calculating the sum of a container's elements or finding a container's maximum value).
- `transform_reduce` performs a transformation on each element in a range or each pair of elements in a pair of ranges, then reduces the results to a single value.

Parallel Algorithm Names

Perf Some parallel algorithms have different names from their sequential equivalents. For example, the `reduce` algorithm is the parallel equivalent of the `accumulate` algorithm.[28] Unlike `accumulate`, **`reduce` does not guarantee the order in which elements are processed, which enables `reduce` to be parallelized for better performance.**[29]

28. Dietmar Kuhl, "C++17 Parallel Algorithms," October 28, 2017. Accessed February 7, 2022. https://www.youtube.com/watch?v=Ve8cHE9LNfk.

29. Sy Brand, "`std::accumulate` vs. `std::reduce`," May 15, 2018. Accessed February 7, 2022. https://blog.tartanllama.xyz/accumulate-vs-reduce/.

Restrictions

Err Unless the algorithm's documentation specifies otherwise, the functions, lambdas or function objects you pass to algorithms must not modify any objects referred to directly or indirectly by their arguments.[30] Also, the standard library algorithms might copy their function-object arguments, so function objects passed to standard library algorithms should not maintain internal state information that could be incorrect if the function objects are copied.[31]

30. C++ Standard, "25.3.2 Requirements on User-Provided Function Objects." Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/algorithms.parallel#user.

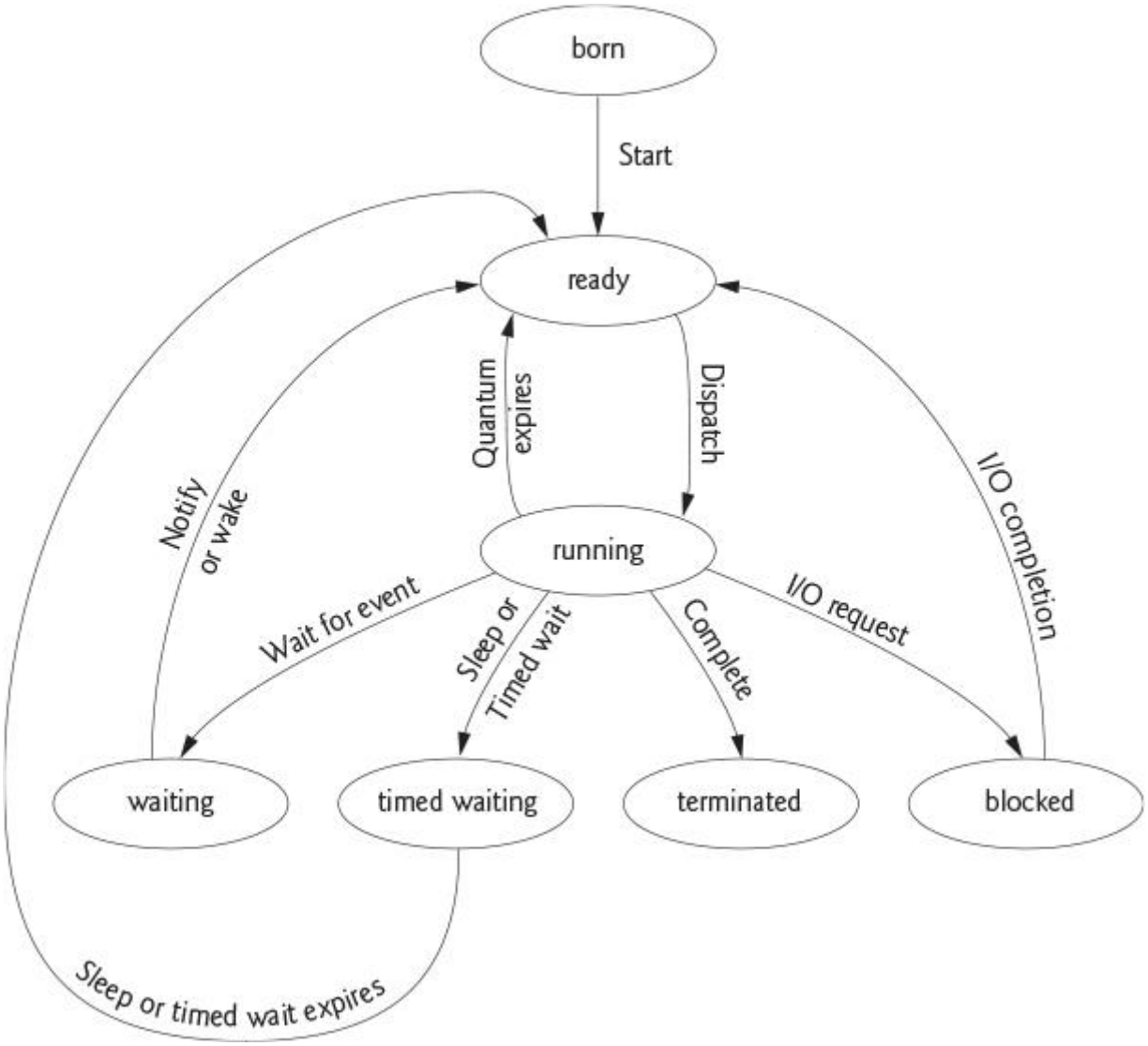
17.3 Multithreaded Programming

Perf In multi-core systems, the hardware can put multiple processors to work truly simultaneously on different parts of your task, enabling your program to complete faster. To take full advantage of multi-core architecture, you need to write multithreaded applications. When a program's tasks are split into separate threads, a multi-core system can run those threads in parallel when sufficient cores are available.

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other applications and other activities the operating system runs on your behalf. All kinds of tasks are typically running in the background on your system. When you run our examples, the time to perform each task will vary based on your computer's processor speed, the number of cores and what's running on your computer. It's not unlike driving to the supermarket. The time it takes can vary based on traffic conditions, weather, presence of emergency vehicles and other factors. Some days the drive might take 10 minutes, but it could take longer, for example, during rush hour or bad weather.

17.3.1 Thread States and the Thread Life Cycle

At any time, a thread is said to be in one of several **thread states**. Here's a general-purpose thread-state switching diagram we borrowed from our operating systems book:[32]



31. C++ Standard, “25.2 Algorithms Requirements.” Accessed February 7, 2022. https://timsongcpp.github.io/cppwp/n4861/algorithms.requirements#10.

SE This diagram should give you a sense of the kinds of state-switching operations going on “under the hood.” Several of these terms are discussed later. C++’s concurrency primitives hide much of this complexity, greatly simplifying multithreaded programming and making it less error-prone.

Born and Ready States

A new thread begins its life cycle in the **born state** and remains in this state until the program starts the thread, which moves it in the **ready state**. In C++, constructing a thread object with a function as an argument (shown in Section 17.4) creates a thread and immediately starts it, making it **ready** to begin performing the task represented by that function.

Running State

Perf A **ready** thread enters the **running state** (i.e., begins executing) when the operating system assigns it to a processor—also known as **dispatching the thread**. Operating systems give each thread a small amount of processor time—called a **quantum** or **timeslice**—to make progress on its task. Deciding how large the quantum should be is a key topic in operating systems design. When its quantum expires, the thread returns to the **ready state**, and the operating system assigns another thread to the processor. Transitions between the **ready** and **running states** are handled solely by the operating system. The process an operating system uses to determine which thread to dispatch and

when is called **thread scheduling**. Scheduling decisions must be made carefully to ensure good performance and avoid problems like **indefinite postponement** of waiting threads (discussed in Section 17.3.2).

Waiting State

Sometimes a **running** thread transitions to the **waiting state** to wait for another thread to perform a task. A **waiting** thread transitions back to the **ready state** when another thread **notifies it to continue executing**.

Timed Waiting State

A **running** thread can enter the **timed waiting state** for a specified time interval. It transitions back to the **ready state** when that time interval expires or the event it's waiting for occurs. **Timed waiting** threads and **waiting** threads cannot use a processor, even if one is available.

A **running** thread can transition to the **timed waiting state** if it provides an optional time interval when waiting for another thread to perform a task. Such a thread returns to the **ready state** when notified by another thread or its wait interval expires.

Perf Putting a **runnable** thread to sleep also transitions the thread to the **timed waiting state**. A **sleeping thread** remains in that state for a period of time (called a **sleep interval**), after which it returns to the **ready state**. Threads sleep when they momentarily do not have work to perform. For example, a word processor may contain a thread that periodically saves the current document. If the thread did not sleep between successive backups, it would require a loop that continually tests whether to save the document. This would consume processor time without performing productive work, reducing system performance. In this case, it's more efficient for the thread to specify a sleep interval equal to the period between successive saves and enter the **timed waiting state**. This thread returns to the **ready state** when its sleep interval expires, at which point it saves the document and reenters the **timed waiting state**.

32. Harvey Deitel, Paul Deitel and David Choffnes, "Chapter 4, Thread Concepts." *Operating Systems, 3/e*, p. 153. Upper Saddle River, NJ: Prentice Hall, 2004.

Blocked State

A **running** thread transitions to the **blocked state** when it attempts to perform a task that cannot be completed immediately. It must temporarily wait until that task completes. For example, when a thread issues an I/O request, the operating system blocks the thread from executing until the I/O completes. At that point, the **blocked** thread transitions to the **ready state** to resume execution. A **blocked** thread cannot use a processor, even if one is available.

Terminated State

A **running** thread enters the **terminated state** when it completes its task.

Thread Scheduling

As we mentioned, **timeslicing** enables threads to share a processor. Without timeslicing, each thread runs to completion (unless it leaves the **running state** and enters the **waiting** or **timed waiting state**) before other threads get a chance to execute. With timeslicing, even if a thread has not finished

executing when its quantum expires, the operating system takes the processor away from the thread and gives it to the next thread, if one is available.

An operating system's **thread scheduler** determines which thread runs next. One simple thread-scheduler implementation ensures that threads each execute for a quantum in a **round-robin** fashion. This process continues until all threads run to completion.

SE Thread scheduling is platform-dependent. The behavior of a multithreaded program could vary across different C++ implementations, different hardware and different operating systems.

17.3.2 Deadlock and Indefinite Postponement

When a higher-priority thread enters the **ready state**, the operating system generally preempts the **running** thread (an operation known as **preemptive scheduling**). Depending on the operating system, a steady influx of higher-priority threads could postpone—possibly indefinitely—the execution of lower-priority threads. Such **indefinite postponement** is referred to as **starvation**. Operating systems can employ a technique called **aging** to prevent starvation—as a thread waits in the **ready state**, the operating system gradually increases the thread's priority to ensure that it will eventually run.

Deadlock

Another problem related to indefinite postponement is called **deadlock**. A thread is **deadlocked** if it is waiting for a particular event that will not occur.[33, 34] In multithreaded systems, resource sharing is one of the primary goals. When resources are shared among a set of threads, with each thread maintaining **exclusive control** over particular resources allocated to it, deadlocks can develop in which some threads will never be able to complete execution. For example, deadlock occurs when a waiting thread, let's call this *thread1*, cannot proceed because it's waiting (either directly or indirectly) for another thread, let's call this *thread2*, to proceed while simultaneously *thread2* cannot proceed because it's waiting (either directly or indirectly) for *thread1* to proceed. The two threads are waiting for each other, so the actions that would enable each thread to continue execution can never occur. The result can be reduced system throughput and even system failure.

33. S. S. Isloor and T. A. Marsland, "The Deadlock Problem: An Overview," *Computer*, Vol. 13, No. 9, September 1980, pp. 58–78.

34. D. Zobel, "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review*, Vol. 17, No. 4, October 1983, pp. 6–16.

Four Necessary Conditions for Deadlock

Coffman, Elphick and Shoshani[35] proved that the following **four conditions are necessary for deadlock to exist**:

1. A resource may be acquired for the exclusive use of only one thread at a time (**mutual exclusion condition**).
2. A thread that has acquired an exclusive resource may hold that resource while the thread waits to obtain other resources (**wait-for condition**, also called the **hold-and-wait condition**).

3. Once a thread has obtained a resource, the system cannot remove it from the thread's control until the thread has finished using the resource (**no-preemption condition**).
4. Two or more threads are locked in a "circular chain" in which each thread is waiting for one or more resources that the next thread in the chain is holding (**circular-wait condition**).

Because these are necessary conditions, the existence of a deadlock implies that each of them must be in effect, so disallowing any of these necessary conditions prevents deadlocks from occurring. Taken together, all four conditions are necessary and **sufficient** for deadlock to exist (i.e., if they are all in place, the system is deadlocked).

Indefinite Postponement

In **indefinite postponement**, a thread that is not deadlocked could wait for an event that might never occur or might occur unpredictably far in the future because of biases in the system's resource-scheduling policies. In some cases, the price for making a system deadlock-free and indefinite-postponement-free is high. In mission-critical systems, the price must be paid no matter how high because allowing deadlock or indefinite postponement to develop could be catastrophic, especially if it puts human life at risk.

Preventing Deadlock

There are various deadlock-prevention techniques. Havender observed that a deadlock cannot occur if a system prevents any of the **four necessary conditions** and suggested several deadlock-prevention strategies.[36]

The most practical approach is for each thread to request all its required resources at once and not proceed until all have been granted. If a thread requests and gets all its needed resources at once, runs to completion, then releases them, there cannot be a circular wait, and the thread cannot deadlock. If the thread cannot get all its resources at once, it should cancel the request and try again later, allowing other threads to proceed.

35. E. G. Coffman, Jr., M. J. Elphick and A. Shoshani, "System Deadlocks," *Computing Surveys*, Vol. 3, No. 2, June 1971, p. 69.

36. J. W. Havender, "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, Vol. 7, No. 2, 1968, pp. 74–84.

Err Perf Requesting all required resources at once is not a perfect solution—it can lead to indefinite postponement and might result in poor system-resource utilization. For example, if a thread performs a 10-minute task requiring five resources,

- one for the task's entire duration and
- the others for only the task's last minute of execution,

you'll get poor utilization on the four you do not need until the last minute.

The Dangers of Waiting

Deadlock and indefinite postponement each involve some form of waiting. In concurrent programming, these waiting scenarios often develop in subtle ways that are not easily detectable, especially as the number of active concurrent tasks grows. The consequences when people's lives are

at stake could be devastating. As you work with concurrency, you should develop a healthy sense of caution. Can you build correctly functioning concurrent systems? Yes, you can. Is it easy? Not always. A crucial key to building reliable business-critical and mission-critical systems is the trend toward developing and employing higher-level concurrency primitives. A significant part of this chapter is devoted to this higher-level approach, but first, we'll look at a few low-level building blocks.

17.4 Launching Tasks with `std::jthread`

11 20 The C++ standard provides two classes for launching concurrent tasks in an application—`std::thread` (C++11) and `std::jthread` (C++20).[37] Both classes are defined in the `<thread>` header, which also includes utility functions, such as `get_id`, `sleep_for` and `sleep_until`. We use `std::jthread`[38, 39] exclusively in this chapter because it fixes several problems with `std::thread`.

37. We'll also present higher-level capabilities in which you do not explicitly create threads.

38. In `clang++`, versions 13 and higher support `std::jthread`. To run examples that use it on earlier `clang++` versions, replace `std::jthread` with `std::thread` and `join` the threads as shown in Section 17.4.3.

39. On Linux, add the `-pthread` compiler flag to your compilation commands to use `std::jthread`.

To specify a task that can execute concurrently with other tasks:

- create a function, lambda or function object that defines the task to perform, then
- initialize a `std::jthread` object with the function, lambda or function object to execute it in a separate thread.

The task's return value is ignored. As you'll see, other mechanisms are used to communicate data between threads.

Exceptions in `jthreads`

Err If a `std::jthread`'s task exits via an exception, the program terminates by calling `std::terminate`.

17.4.1 Defining a Task to Perform in a Thread

This example consists of the header `printtask.h` (Fig. 17.3) and the `main` application (Fig. 17.4). We use the function `id` (Fig. 17.3, lines 11–15) in output statements to show which thread is executing at a given time. Every thread has a **unique ID number**, including `main`'s thread, threads created with `std::jthread` or `std::thread` and threads created for you by library functions. In line 13, the expression

```
std::this_thread::get_id()
```

invokes the `std::this_thread` namespace's `get_id` function to get the currently executing thread's unique ID number, which is returned as a `std::thread::id` object. We use that object's overloaded `operator<<` to convert the ID to a `std::string`.

```
1  // Fig. 17.3: printtask.h
2  // Function printTask defines a task to perform in a separate thread.
```

```
 3   #include <chrono>
 4   #include <fmt/format.h>
 5   #include <iostream>
 6   #include <sstream>
 7   #include <string>
 8   #include <thread>
 9
10   // get current thread's ID as a string
11   std::string id() {
12      std::ostringstream out;
13      out << std::this_thread::get_id();
14      return out.str();
15   }
16
17   // task to perform in a separate thread
18   void printTask(const std::string& name,
19      std::chrono::milliseconds sleepTime) {
20
21      // <chrono> library features we'll use for timing
22      using std::chrono::steady_clock;
23      using std::chrono::duration_cast;
24      using std::chrono::milliseconds;
25
26      std::cout << fmt::format("{} (ID {}) going to sleep for {} ms\n",
27         name, id(), sleepTime.count());
28
29      auto startTime{steady_clock::now()}; // get current time
30
31      // put thread to sleep for sleepTime milliseconds
32      std::this_thread::sleep_for(sleepTime);
33
34      auto endTime{steady_clock::now()}; // get current time
35      auto time{duration_cast<milliseconds>(endTime - startTime)};
36      auto difference{duration_cast<milliseconds>(time - sleepTime)};
37      std::cout << fmt::format("{} (ID {}) awakens after {} ms ({} + {})\n",
38         name, id(), time.count(), sleepTime.count(), difference.count());
39   }
```

Fig. 17.3 Function `printTask` defines a task to perform in a separate thread.

Function `printTask` (lines 18–39) implements the task we'd like to perform. The parameters represent

- a `name` we use to identify each task in the program's output and
- a `sleepTime` in milliseconds that we use to force the executing thread to give up the processor for at least that amount of time.

When a thread calls `printTask`:

Lines 26–27 display a message indicating the currently executing task's `name`, the **unique thread ID** and the `sleepTime` in milliseconds.[40]

40. When multiple threads display output with `std::cout`, their outputs might be interleaved, making the outputs appear corrupted. Preventing this requires thread synchronization (Section 17.6).

Line 29 gets the time before the thread goes to sleep.

Line 32 invokes the `std::this_thread` namespace's `sleep_for` function to put the thread to sleep for at least the specified amount of time. At this point, the thread loses the processor, possibly enabling another thread to execute. **Our examples often make threads sleep to simulate performing work. We also use randomized sleeping to emphasize that you cannot predict when each thread will receive**

processor time to execute its task. The `std::this_thread` namespace also provides `sleep_until`, which sleeps until a specified time.
When the thread's sleep time expires, **the thread reenters the ready state but does not necessarily start executing immediately**. Eventually, when the operating system assigns a processor to the thread, line 34 gets the time, line 35 calculates the total time the thread was not executing, and line 36 calculates the difference between the total time and the `sleepTime`. Then, lines 37–38 display the times. When `printTask` terminates, its `jthread` enters the **terminated state**.

17.4.2 Executing a Task in a `jthread`

Figure 17.4 launches two concurrent threads that execute `printTask` (Fig. 17.3) and shows two sample outputs. Lines 14–16 set up random-number generation for choosing random sleep times of up to 5,000 milliseconds. Line 18 creates a `vector` to store `std::jthreads`. We use this to enable `main` to wait for the threads to complete their tasks before the program terminates. We'll say more about this momentarily.

```
1  // Fig. 17.4: printtask.cpp
2  // Concurrently executing tasks with std::jthreads.
3  #include <chrono>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <random>
7  #include <string>
8  #include <thread>
9  #include <vector>
10 #include "printtask.h"
11
12 int main() {
13    // set up random-number generation
14    std::random_device rd;
15    std::default_random_engine engine{rd()};
16    std::uniform_int_distribution ints{0, 5000};
17
18    std::vector<std::jthread> threads; // stores the jthreads
19
20    std::cout << "STARTING JTHREADS\n";
21
22    // start two jthreads
23    for (int i{1}; i < 3; ++i) {
24       std::chrono::milliseconds sleepTime{ints(engine)};
25       std::string name{fmt::format("Tasks {}", i)};
26
27       // create a jthread that calls printTask, passing name and sleepTime
28       // as arguments and store the jthread, so it is not destructed until
29       // the vector goes out of scope at the end of main; each jthread's
30       // destructor automatically joins the jthread
31       threads.push_back(std::jthread{printTask, name, sleepTime});
32    }
33
34    std::cout << "\nJTHREADS STARTED\n";
35    std::cout << "\nMAIN ENDS\n";
36 }
```

```
STARTING JTHREADS

JTHREADS STARTED

MAIN ENDS
Task 2 (ID 15704) going to sleep for 4547 ms
Task 1 (ID 15624) going to sleep for 3648 ms
Task 1 (ID 15624) awakens after 3651 ms (3648 + 3)
Task 2 (ID 15704) awakens after 4555 ms (4547 + 8)
STARTING JTHREADS

JTHREADS STARTED

MAIN ENDS
Task 1 (ID 9368) going to sleep for 441 ms
Task 2 (ID 16876) going to sleep for 2614 ms
Task 1 (ID 9368) awakens after 449 ms (441 + 8)
Task 2 (ID 16876) awakens after 2618 ms (2614 + 4)
```

Fig. 17.4 Concurrently executing tasks with `std::jthreads`.

Lines 23–32 create two **`std::jthread`** objects:

- Line 24 picks a **random sleep time**.
- Line 25 creates a `string` that we'll use to identify the task in the outputs.
- Line 31 creates each **`jthread`** and appends it to the `vector`. The **`jthread`** constructor receives the function to execute (`printTask`) and the arguments that should be passed to that function (`name` and `sleepTime`). This statement creates the **`jthread`** as a **temporary object**, so the `vector`'s `push_back` function that receives an *rvalue* reference moves the object into the new `vector` element.

Constructing a **`jthread`** with a function to execute starts the **`jthread`**, so the operating system can schedule it for execution on one of the system's cores. Line 34 outputs a message indicating the threads were started, and line 35 indicates that `main` ends.

Waiting for Previously Scheduled Tasks to Terminate

11 After scheduling tasks to execute, you typically want to **wait for them to complete**—for example, to use their results. You tell `main` to wait for a **`jthread`** to complete its task by "**joining the thread**" with a call to its **`join` function**. This can be done explicitly or, as we do in this program, **implicitly via `jthread`'s destructor**—one of **`jthread`**'s benefits over C++11's **`std::thread`** (see Section 17.4.3). As you know, when `main` ends, its local variables go out of scope, and their destructors are called. When the `vector`'s destructor executes, each **`jthread`** element's destructor executes, calling that **`jthread`**'s **`join`** function. When all the joined **`jthreads`** have completed their execution, the program can terminate.

Main Thread

The code in `main` executes in the **main thread**. Function `printTask` executes when the operating system dispatches the corresponding **`jthread`**, sometime after it enters the **ready state**.

Sample Outputs

The sample outputs show each task's name and sleep time as the thread goes to sleep. **The thread with the shortest sleep time typically awakens first, but that is not guaranteed.** When each thread continues, it displays its task name, **unique thread ID** and timings, then terminates. The first output

shows the tasks going to sleep in a different order from that in which we created their `jthread`s. **We cannot predict the order in which the tasks will start executing, even if we know the order in which they were created and started**. This is one of the challenges of multithreaded programming.

17.4.3 How `jthread` Fixes `thread`

CG Err **The C++ Core Guidelines say to prefer `std::jthread` over `std::thread`.**[41] Class `thread` has various problems. As discussed in the preceding example, a `jthread` automatically **`join`s each thread** to ensure that its thread completes execution before the program terminates. In fact, the name `jthread` is short for "**joining thread**." On the other hand, **if you do not join a `thread` or detach it before it's destroyed, its destructor calls `std::terminate`, immediately terminating the application.**[42] To prevent that, you must explicitly `join` each `thread`. If the previous program used `thread`s, we would have added a loop like the following before `main`'s closing brace:

41. C++ Core Guidelines, "CP.25: Prefer `gsl::joining_thread` over `std::thread`." Accessed February 7, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-join-ing_thread. [**Note:** This guideline says to prefer `std::jthread` in C++20.]

42. **Detaching a thread** separates it from the `thread` or `jthread`, but the operating system continues executing the thread. **The C++ Core Guidelines recommend against detaching** because it "makes it harder to monitor and communicate with the detached thread." For more details, see "CP.26: Don't `detach()` a Thread." Accessed February 10, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-detached_thread.

```
for (auto& t : threads) {
  t.join();
}
```

Err Even with the preceding loop, there's another potential error. If the function that launched the `thread`s exits via an uncaught exception before joining each thread, the `thread`s will be destroyed, and the first `thread` destructor to execute will call `std::terminate`.

Typically, you should always `join` **each thread**, so `jthread` fixes the preceding problem by calling its own `join` function in its destructor.[43] Of course, the destructor executes anytime a `jthread` goes out of scope:

- at the end of the block that created the `jthread` or
- when the block terminates due to an exception.

Class `jthread` also fixes other problems with `thread`. In particular, `jthread` supports **cooperative cancellation** (Section 17.9), supports proper move semantics, and is an **RAII type** (discussed in Section 11.5) that correctly cleans up the resources it uses.[44]

17.5 Producer–Consumer Relationship: A First Attempt

Err As you'll soon see, **when concurrent threads share mutable data and that data is modified by one or more of them, indeterminate results may occur**:

- If one thread is in the process of updating a shared object and another thread also tries to update it, it's uncertain which thread's update will take effect.
- Similarly, if one thread is in the process of updating a shared object and another thread tries to read it, it's uncertain whether the reading thread will see the old value or the new one.

In such cases, the program's behavior cannot be trusted. Sometimes the program will produce the correct results, and sometimes it will not. There won't be any indication that the shared mutable object was manipulated incorrectly. **Worse yet, multithreaded code could appear to run correctly on one run and incorrectly on the next.**

In this section and the next, we'll present two examples. The first demonstrates the problems with concurrent threads **accessing shared mutable data**. The second fixes those problems. Both demonstrate the **producer–consumer relationship** in which

- a **producer** thread generates data and **stores it in a shared object** and
- a **consumer** thread **reads data from that shared object**.

Another Example of a Producer–Consumer Relationship

Print spooling is a common example of a producer–consumer relationship. A printer is an **exclusive resource**. Although a printer might not be available when you want to print from an application (the producer), you can still "complete" the print task. The data is temporarily stored (called **spooling**) until the printer becomes available. Similarly, when the printer (a consumer) is available, it doesn't wait until a current user wants to print. The spooled print jobs can be printed when the printer becomes available.

43. Nicolai Josuttis, "Why and How We Fixed `std::thread` by `std::jthread`," July 29, 2020. Accessed February 7, 2022. https://www.youtube.com/watch?v=elFil2VhlH8.
44. Nicolai Josuttis, *C++20: The Complete Guide*," Chapter 15, `std::jthread` and Stop Tokens," November 21, 2021. http://cppstd20.com/.

Synchronization

In a multithreaded producer–consumer relationship, a **producer thread** generates data, placing it in a shared object called a **buffer**. A **consumer thread** reads data from the buffer. This relationship requires **synchronization** to ensure that values are produced and consumed correctly. All operations on **shared mutable data** accessed by concurrent threads must be **guarded with a lock** to prevent corruption, as we'll show in Section 17.6.

State Dependence

Operations on the shared buffer also are **state-dependent**. The operations should proceed only if the buffer is in the correct state:

- If the buffer is in a **not-full state**, the producer may produce.
- If the buffer is in a **not-empty state**, the consumer may consume.

Similarly:

- If the buffer is in the **full state** when the producer wants to write a new value, it must **wait** until there's space in the buffer.
- If the buffer is in the **empty state** when the consumer wants to read a value, it must **wait** for new data to become available.

Logic Errors from Lack of Synchronization

Err Let's illustrate the dangers of concurrent threads sharing mutable data **without proper synchronization**. In Figs. 17.5–17.6, a producer thread writes the numbers 1 through 10 into a shared buffer—in this case, an `int` variable called `m_buffer` in line 23 of Fig. 17.5. A consumer thread reads this data from the shared buffer and displays the data. Though this might seem harmless, this program's output will show the errors that can occur as the unrestrained producer writes (produces) values into the shared buffer at will and the unrestrained consumer reads (consumes) those values from the shared buffer at will.

Each value the producer thread writes to the shared buffer must be consumed **exactly once** by the consumer thread. The threads in this example are not synchronized, so

- Err **data can be lost or garbled** if the producer places new data into the shared buffer before the consumer reads the previous data, and
- Err **data can be incorrectly duplicated** if the consumer consumes the same data again before the producer produces the next value.

To show these possibilities, the consumer thread in the following example keeps a total of all the values it reads. The producer thread produces values from 1 through 10. If the consumer correctly reads each value produced once and only once, the total will be 55. As you'll see, the unsynchronized producer–consumer pair can create incorrect totals (other than 55). Interestingly, the unsynchronized producer–consumer pair can incorrectly create the correct total of 55. We, of course, want to ensure correct operation. We'll show how to do that in Section 17.6.

`UnsynchronizedBuffer`

In Fig. 17.6, we'll share an object of class `UnsynchronizedBuffer` (Fig. 17.5) between the concurrent producer and consumer threads. `UnsynchronizedBuffer` maintains a single `int` data member (line 22). The producer thread will call the class's **`put`** function (lines 11–14) to place a value in the `int`, and the consumer thread will call the class's **`get`** function (lines 17–20) to retrieve the `int`'s value.

```
1  // Fig. 17.5: UnsynchronizedBuffer.h
2  // UnsynchronizedBuffer incorrectly maintains a shared integer that is
3  // accessed by a producer thread and a consumer thread.
4  #pragma once
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <string>
8
9  class UnsynchronizedBuffer {
10 public:
11    // place value into buffer
12    void put(int value) {
13       std::cout << fmt::format("Producer writes\t{:2d}"xs, value);
14       m_buffer = value;
15    }
16
17    // return value from buffer
18    int get() const {
19       std::cout << fmt::format("Consumer reads\t{:2d}", m_buffer);
```

```
20          return m_buffer;
21       }
22    private:
23       int m_buffer{-1}; // shared by producer and consumer threads
24    };
```

Fig. 17.5 `UnsynchronizedBuffer` incorrectly maintains a shared integer that is accessed by a producer thread and a consumer thread. This class is not thread-safe.

This Buffer Is Not Protected by Synchronization

Class `UnsynchronizedBuffer` does not synchronize access to its data, so it is **not thread-safe**. Line 23 initializes the `UnsynchronizedBuffer`'s `m_buffer` member to `-1`. We use this value to show the case in which the **consumer thread attempts to consume a value before the producer thread ever places a value in `m_buffer`**. Function **`put`** simply assigns its argument to `m_buffer` (line 14), and function **`get`** simply returns `m_buffer`'s value (line 20).

Main Application

In Fig. 17.6, line 12 creates the `UnsynchronizedBuffer` object `buffer`, which stores the data shared by the concurrent producer and consumer threads. The producer thread will invoke the lambda `produce` (lines 15–36), and the consumer thread will invoke the lambda `consume` (lines 39–60). We'll discuss each lambda in more detail momentarily. Both lambdas' introducers capture `buffer` by reference, so the concurrent threads executing these lambdas share line 12's `buffer` object. Lines 62–63 display column heads for this program's output. Lines 65–66 create two **`jthread`s**—`producer` executes the `produce` lambda, and `consumer` executes the `consume` lambda. Each goes out of scope at the end of `main`, which **`join`s** the threads. The italicized text in the output is our commentary, which is not part of the program's output.

```
1  // Fig. 17.6: SharedBufferTest.cpp
2  // Application with concurrent jthreads sharing an unsynchronized buffer.
3  #include <chrono>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <random>
7  #include <thread>
8  #include "UnsynchronizedBuffer.h"
9
10 int main() {
11    // create UnsynchronizedBuffer to store ints
12    UnsynchronizedBuffer buffer;
13
14    // lambda expression that produces the values 1-10 and sums them
15    auto produce{
16       [&buffer]() {
17          // set up random-number generation
18          std::random_device rd;
19          std::default_random_engine engine{rd()};
20          std::uniform_int_distribution ints{0, 3000};
21
22          int sum{0};
23
24          for (int count{1}; count <= 10; ++count) {
25             // get random sleep time then sleep
```

```
26              std::chrono::milliseconds sleepTime{ints(engine)};
27              std::this_thread::sleep_for(sleepTime);
28
29              buffer.put(count); // set value in buffer
30              sum += count; // add count to sum of values produced
31              std::cout << fmt::format("\t{:2d}\n", sum);
32           }
33
34           std::cout << "Producer done producing\nTerminating Producer\n";
35        }
36     };
37
38     // lambda expression that consumes the values 1-10 and sums them
39     auto consume{
40        [&buffer]() {
41           // set up random-number generation
42           std::random_device rd;
43           std::default_random_engine engine{rd()};
44           std::uniform_int_distribution ints{0, 3000};
45
46           int sum{0};
47
48           for (int count{1}; count <= 10; ++count) {
49              // get random sleep time then sleep
50              std::chrono::milliseconds sleepTime{ints(engine)};
51              std::this_thread::sleep_for(sleepTime);
52
53              sum += buffer.get(); // get buffer value and add to sum
54              std::cout << fmt::format("\t\t\t{:2d}\n", sum);
55           }
56
57           std::cout << fmt::format("\n{} {}\n{}\n",
58              "Consumer read values totaling", sum, "Terminating Consumer");
59        }
60     };
61
62     std::cout << "Action\t\tValue\tSum of Produced\tSum of Consumed\n";
63     std::cout << "------\t\t-----\t---------------\t---------------\n";
64
65     std::jthread producer{produce}; // start producer jthread
66     std::jthread consumer{consume}; // start consumer jthread
67  }
```

```
Action          Value   Sum of Produced Sum of Consumed
------          -----   --------------- ---------------
Producer writes   1       1
Producer writes   2       3               -1 lost
Consumer reads    2                       2
Consumer reads    2                       4 -2 read again
Producer writes   3       6
Consumer reads                            7
Producer writes   4      10
Producer writes   5      15               -4 lost
Consumer reads    5                      12
Producer writes   6      21
```

```
Producer writes   7      28                 –6 lost
Consumer reads    7                      19
Consumer reads    7                      26 –7 read again
Consumer reads    7                      33 –7 read again
Producer writes   8      36
MaProducer writes 9      45                 –8 lost
Consumer reads    9                      42
Producer writes  10      55
Producer done producing
Terminating Producer
Consumer reads   10                      52
Consumer reads   10                      62–10 read again

Consumer read values totaling 62
Terminating Consumer
Action          Value  Sum of Produced Sum of Consumed
------          -----  --------------- ---------------
Consumer reads  -1                     -1 –reads -1 bad data (producer must go first)
Producer writes  1      1
Producer writes  2      3                  –1 lost
Consumer reads   2                            1
Consumer reads   2                      3 –2 read again
Producer writes  3      6
Consumer reads   3                      6
Producer writes  4     10
Consumer reads   4                     10
Consumer reads   4                     14 –4 read again
Producer writes  5     15
Consumer reads   5                     19
Producer writes  6     21
Consumer reads   6                     25
Producer writes  7     28
Consumer reads   7                     32
Producer writes  8     36
Producer writes  9     45                 –8 lost
Producer writes 10     55                 –9 lost
Producer done producing
Terminating Producer
Consumer reads  10                     42

Consumer read values totaling 42
Terminating Consumer
Action          Value  Sum of Produced Sum of Consumed
------          -----  --------------- ---------------
Producer writes  1      1
Consumer reads   1                      1
Producer writes  2      3
Producer writes  3      6                  –2 lost
Consumer reads   3                      4
Producer writes  4     10
Consumer reads   4                      8
Consumer reads   4                     12 –4 read again
Consumer reads   4                     16 –4 read again
Consumer reads   4                     20 –4 read again
Producer writes  5     15
```

```
Producer writes  6     21                    -5 lost
Consumer reads   6                      26
Producer writes  7     28
Producer writes  8     36                    -7 lost
Producer writes  9     45                    -8 lost
Consumer reads   9                      35
Producer writes 10     55
Producer done producing
Terminating Producer
Consumer reads  10                     45
Consumer reads  10                     55 -10 read again

Consumer read values totaling 55             -Accidentally correct total
Terminating Consumer
```

Fig. 17.6 Application with concurrent `jthreads` sharing an unsynchronized buffer. (*Caution:* `UnsynchronizedBuffer` is *not* thread-safe.)

Producer Thread

The `producer` thread executes the `produce` lambda (lines 15–36). Each loop iteration **sleeps** (line 27) for 0 to 3,000 milliseconds. When the **thread awakens**, line 29 sets the shared `buffer`'s value by passing `count` to the object's `put` function. Lines 30–31 keep a total of the values produced so far and display that total. When the loop completes, line 34 indicates that the `producer` has finished producing data and is terminating. Then the lambda finishes executing, and the **`producer` thread terminates**. Any function called from a **`jthread`'s task**, such as `buffer`'s `put` function, executes in that thread. This fact will be important in Sections 17.6–17.7 when we **synchronize the producer–consumer relationship**.

Consumer Thread

The **`consumer` thread** executes the `consume` lambda (lines 39–60). Lines 48–55 iterate 10 times. Each iteration **sleeps** (line 51) for a random time interval of 0 to 3,000 milliseconds. Next, line 53 uses the `buffer`'s `get` function to retrieve the `buffer`'s value, then adds it to `sum`. Line 54 displays the total of the values consumed so far. When the loop completes, lines 57–58 display the sum of the consumed values, the lambda finishes executing, and the **`consumer` thread terminates**. Once both threads terminate, the program ends.

Random-Number Generation

11 **C++11's random-number generation is not thread-safe**. To ensure that each thread can safely produce random numbers, we defined **separate random-number generators** in the lambdas `produce` (lines 18–20) and `consume` (lines42–44), rather than sharing one random-number generator between them.[45]

45. C++ Standard, "16.5.5.10 Data Race Avoidance." Accessed February 7, 2022. https://timsongcpp.github.io/cppwp/n4861/res.on.data.races.

We Call `std::this_thread::sleep_for` Only for Demonstration Purposes

Throughout this chapter, we'll refer to **asynchronous concurrent threads**. When we say "asynchronous," we mean that the threads work pretty much independently of one another.[46] To emphasize that **you cannot predict the relative speeds of asynchronous concurrent threads**, we call function **`std::this_thread::sleep_for`** in the `produce` and `consume` lambdas. Thus, we do not know when the producer thread will write a new value or when the consumer thread will read a value. **In multithreaded applications, it's generally unpredictable when and for how long each thread will**

perform its task when it has a processor. These thread-scheduling issues are controlled by the operating system.
46. Harvey Deitel, Paul Deitel and David Choffnes, Chapter 3, "Process Concepts." *Operating Systems, 3/e*, p. 124. Upper Saddle River, NJ: Prentice Hall, 2004.

Without the `sleep_for` calls, and if the `producer` were to execute first, given today's processor speeds, the `producer` would likely complete its task before the `consumer` got a chance to execute. If the `consumer` were to execute first, it would likely consume the same garbage data ten times, then terminate before the `producer` could produce the first real value.

Analyzing the Outputs[47]

47. As in Fig. 17.4, this program's outputs can be interleaved in a manner that makes the output appear corrupted. Preventing this requires thread synchronization, which we discuss in Section 17.6.

Err Recall that **the `producer` thread should execute first**, and **every value it produces should be consumed exactly once by the `consumer` thread**. We highlighted lines in the output where the `producer` or `consumer` acted out of order to emphasize the problems caused by **failure to synchronize access to shared mutable data**:

- In the first output of Fig. 17.6, notice that the `producer` writes `1` and `2` before the `consumer` reads its first value (`2`). Therefore, the value `1` is **lost**. Later, `4`, `6` and `8` are **lost**, while `2` and `10` are **read twice**, and `7` is **read three times**. So the first output produces an incorrect total of 62 instead of the correct total of 55.
- In the second output, the `consumer` reads the garbage value `-1` **before** the `producer` ever writes a value. Meanwhile, `1`, `8` and `9` are all **lost**, and `2` and `4` are **read twice**. The result is an incorrect consumer total of 42.
- In the third output, you see that **it's possible for the `consumer` to read values that accidentally total 55 without reading every value from `1` through `10` exactly once**. In this case, the values `2`, `5`, `7` and `8` are all **lost**, the value `4` was read **four times**, and the value `10` was **read twice**.

The outputs clearly show that access to a shared mutable object by concurrent threads must be controlled carefully; otherwise, a program will likely produce incorrect results—and perhaps even worse, could accidentally produce a "correct result" for the wrong reasons.

Err One challenge of multithreaded programming is spotting errors. They may occur so infrequently and unpredictably that a broken program does not produce incorrect results during testing, creating the illusion that it's correct. This is why you should **use predefined containers and higher-level primitives that handle the synchronization for you**.

SE To solve the problems of **lost** and **duplicated** data, the next section presents an example in which we **synchronize access to the shared object, guaranteeing that each value will be processed once and only once**.

17.6 Producer–Consumer: Synchronizing Access to Shared Mutable Data

The errors in Section 17.5's program can be attributed to the fact that **`Unsynchronized-Buffer` is not thread-safe**. It allowed uncoordinated concurrent producer and consumer threads to modify and

read shared mutable data, which leads to **data races** (also called **race conditions**).[48, 49] The thread that "wins the race" by getting there first performs its task, even if it should not. There's no guarantee of the order in which the concurrent producer and consumer threads will perform their tasks, resulting in cases in which:

48. C++ Standard, "Data Races." Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/intro.races#def:data_race.

49. Arthur O'Dwyer, "Back to Basics: Concurrency," October 6, 2020. Accessed February 7, 2022. https://www.youtube.com/watch?v=F6Ipn7gCOsY. If you enjoy watching videos of experts making one-hour conference presentations, check out this video. O'Dwyer heads the CppCon committee responsible for the "Back to Basics" track. He and other experts offer sessions on a broad range topics important to C++ developers.

- the producer overwrites previously written values before they're consumed, causing lost data, or
- the consumer reads invalid data (`-1`) before the producer has produced its first legitimate value or
- the consumer reads **stale values** it read previously.

The C++ standard emphasizes that "**a memory location cannot be safely accessed by two threads without some form of locking unless they are both read accesses.**"[50]

50. "C++11 Language Extensions—Concurrency." Accessed February 7, 2022. https://isocpp.org/wiki/faq/cpp11-language-concurrency.

Thread Synchronization, Mutual Exclusion and Critical Sections

The problems from the last example can be solved by **giving only one thread at a time exclusive access to code that manipulates the shared mutable data**. During that time, the other thread must **wait**. When the thread with exclusive access finishes accessing the shared mutable data, **the waiting thread can proceed**. This **thread synchronization** process coordinates access to shared mutable data by concurrent threads. Each thread accessing a shared object **excludes** the other thread from doing so simultaneously. This is called **mutual exclusion**. The code sections that we protect using mutual exclusion are called **critical sections**.

Executing a Set of Operations As If They Are One Operation

To make our shared buffer thread-safe, we must ensure that **only one thread at a time can store a value into the buffer or read a value from the buffer**. We must also ensure that these operations cannot be divided into smaller suboperations—known as making the operations **atomic**. We do this by allowing only one thread to execute `put` or `get` at a time:

- If the producer is executing `put` when the consumer tries to execute `get`, the consumer must **wait** until the producer finishes its `put` call.
- Similarly, if the consumer is executing `get` when the producer tries to execute `put`, the producer must **wait** until the consumer finishes its `get` call.

Immutable Data Does Not Require Synchronization

CG The synchronization we demonstrate in the next example is required only for **shared mutable data**, which might change during its lifetime. **Immutable data** does not change, so any number of concurrent threads can access the data. The **C++ Core Guidelines say, "You can't have a race condition on a constant,"** and for this reason indicate that you should

- **"use `const` to define objects with values that do not change after construction"** and
- **"use `constexpr` for values that can be computed at compile time."**[51]

51. C++ Core Guidelines, "Con: Constants and Immutability." Accessed February 7, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-const.

CG Doing so indicates the variables' values will not change after they're initialized, preventing accidental modification that could compromise thread safety. The C++ Core Guidelines also recommend **minimizing the use of shared mutable data to avoid data races.**[52]

52. C++ Core Guidelines, "CP.3: Minimize Explicit Sharing of Writable Data." Accessed February 7, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-data.

17.6.1 Class `SynchronizedBuffer`: Mutexes, Locks and Condition Variables

Figures 17.7–17.8 demonstrate a producer thread and a consumer thread correctly accessing a **synchronized** shared mutable buffer. In this example:

- the producer always produces a value first,
- the consumer correctly consumes only after the producer produces a value and
- the producer correctly produces the next value only after the consumer consumes the previous (or first) value.

Perf SE As you'll see, `SynchronizedBuffer`'s **`put`** and **`get`** functions (Fig. 17.7) handle the synchronization. We output messages from these functions for demonstration purposes only. I/O is slow compared to processor operations. So, I/O should not be performed in critical sections because it's crucial to minimize the amount of time that an object is "locked."

```
// Fig. 17.7: SynchronizedBuffer.h
// SynchronizedBuffer maintains synchronized access to a shared mutable
// integer that is accessed by a producer thread and a consumer thread.
#pragma once
#include <condition_variable>
#include <fmt/format.h>
#include <mutex>
#include <iostream>
#include <string>

using namespace std::string_literals;

class SynchronizedBuffer {
public:
   // place value into m_buffer
   void put(int value) {
      // critical section that requires a lock to modify shared data
      {
         // lock on m_mutex to be able to write to m_buffer
         std::unique_lock dataLock{m_mutex};

         if (m_occupied) {
            std::cout << fmt::format(
               "Producer tries to write.\n{:<40}{}\t\t{}\n\n",
```

```
25                   "Buffer full. Producer waits.", m_buffer, m_occupied);
26
27                // wait on condition variable m_cv; the lambda in the second
28                // argument ensures that if the thread gets the processor
29                // before m_occupied is false, the thread continues waiting
30                m_cv.wait(dataLock, [&]() {return !m_occupied;});
31             }
32
33             // write to m_buffer
34             m_buffer = value;
35             m_occupied = true;
36
37             std::cout << fmt::format("{:<40}{}\t\t{}\n\n",
38                "Producer writes "s + std::to_string(value),
39                m_buffer, m_occupied);
40          } // dataLock's destructor releases the lock on m_mutex
41
42          // if consumer is waiting, notify it that it can now read
43          m_cv.notify_one();
44       }
45
46       // return value from m_buffer
47       int get() {
48          int value; // will store the value returned by get
49
50          // critical section that requires a lock to modify shared data
51          {
52             // lock on m_mutex to be able to read from m_buffer
53             std::unique_lock dataLock{m_mutex};
54
55             if (!m_occupied) {
56                std::cout << fmt::format(
57                   "Consumer tries to read.\n{:<40}{}\t\t{}\n\n",
58                   "Buffer empty. Consumer waits.", m_buffer, m_occupied);
59
60                // wait on condition variable m_cv; the lambda in the second
61                // argument ensures that if the thread gets the processor
62                // before m_occupied is true, the thread continues waiting
63                m_cv.wait(dataLock, [&]() {return m_occupied;});
64             }
65
66             value = m_buffer;
67             m_occupied = false;
68
69             std::cout << fmt::format("{:<40}{}\t\t{}\n{}\n",
70                "Consumer reads "s + std::to_string(m_buffer),
71                m_buffer, m_occupied, std::string(64, '-'));
72          } // dataLock's destructor releases the lock on m_mutex
73
74          // if producer is waiting, notify it that it can now write
75          m_cv.notify_one();
76
77          return value;
78       }
79    private:
```

```
80      int m_buffer{-1}; // shared by producer and consumer threads
81      bool m_occupied{false};
82      std::condition_variable m_cv;
83      std::mutex m_mutex;
84   };
```

Fig. 17.7 `SynchronizedBuffer` maintains synchronized access to a shared mutable integer that is accessed by a producer thread and a consumer thread.

`SynchronizedBuffer`'s `m_buffer` and `m_occupied` Data Members

Line 80 defines the `int` data member `m_buffer` into which a producer thread will write data and from which a consumer thread will read data. The `bool` data member `m_occupied` (line 81) indicates whether `m_buffer` currently contains data. We'll use this to help keep track of the shared buffer's state for thread synchronization purposes. Variables `m_occupied` and `m_buffer` are both part of a `SynchronizedBuffer`'s state information, so you must synchronize access to both to ensure that the buffer is **thread-safe**.

`SynchronizedBuffer`'s `std::condition_variable` Data Member

As part of synchronizing access to the buffer, we must ensure that the producer and consumer threads each do their work only when the buffer is in an appropriate state. We need a way to allow the threads to **wait, depending on certain conditions**, which we maintain via `m_occupied`:

- The producer can place a new item in the buffer only if the **buffer is not full**—that is, `m_occupied` is `false`.
- The consumer can read an item from the buffer only if the **buffer is not empty**—that is, `m_occupied` is `true`.

If a given condition is `true`, the corresponding thread may proceed. If it's `false`, the corresponding thread must wait until it becomes `true`.

We also need a way to **let a waiting thread know when conditions have changed** so it can proceed:

- If the consumer is waiting to read and the producer writes a new value into the buffer, the **buffer is now full**, so the producer should **notify the waiting consumer** that it can read that value.
- If the producer is waiting to write and the consumer reads the buffer's current value, the **buffer is now empty**, so the consumer should **notify the waiting producer** that it can write into the buffer.

These wait and notify capabilities are provided by a **`std::condition_variable`**[53] (from header **`<condition_variable>`**). Line 82 defines the `m_cv` data member of this type.

53. "`std::condition_variable`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/condition_variable.

`SynchronizedBuffer`'s `std::mutex` Data Member

A common way to implement **mutually exclusive access** to shared mutable resources is by creating **critical sections**. These **synchronized blocks of code** execute atomically using features from the **`<mutex>` header**. **A `std::mutex` can be owned by only one thread at a time.** A thread that requires exclusive access to a resource must first **acquire a lock** on a **`mutex`**—typically at the beginning of a block of code. Other threads attempting to perform operations that require the same **`mutex`** will be **blocked** until the first thread **releases the lock**—typically at the end of a block of code. At that point, the **blocked** threads may attempt to acquire the lock and proceed with the operation.[54]

54. From a December 6, 2021, email correspondence with Anthony Williams: "There is no requirement on which thread gets the `mutex` lock when a `mutex` is unlocked. ... it is often most efficient

from an OS perspective to allow the thread that has been waiting the **shortest time** to … acquire the `mutex` lock, as its data still might be in cache."

CG By placing operations in a **critical section**, we allow only one thread at a time to acquire the lock and perform the operations. If multiple **critical sections** are synchronized with the same **`mutex`**, only one can execute at a time. Line 83 defines **`mutex`** data member `m_mutex`, which we'll use in conjunction with `m_cv` to **protect the critical sections of code** in `SynchronizedBuffer`'s **`put`** and **`get`** functions that access the class's shared mutable data. For data that requires **mutually exclusive access**, the C++ Core Guidelines recommend defining the data together with the **`mutex`** used to protect it. For example, we will protect `SynchronizedBuffer`'s data members by using a **`mutex`** that's also a data member of `SynchronizedBuffer`.[55]

SE Perf **In multithreaded programs, place all accesses to shared mutable data inside critical sections that synchronize on the same `std::mutex`.** All operations within that critical section represent an **atomic** operation. Promptly release the lock when it's no longer needed.

`SynchronizedBuffer` Member Functions

Functions **`put`** (lines 16–44) and **`get`** (lines 47–78) provide **synchronized access to the shared data members** `m_occupied` and `m_buffer`. Only one thread at a time can enter either of these functions' **critical sections** on a particular `SynchronizedBuffer` object. Variable `m_occupied` is used in **`put`** and **`get`** to determine whether it's the producer's turn to write or the consumer's turn to read:

- If `m_occupied` is `false`, `m_buffer` is empty, so the producer can place a value into `m_buffer`, but the consumer cannot read `m_buffer`'s value.
- If `m_occupied` is `true`, `m_buffer` is full, so the consumer can read `m_buffer`'s value, but the producer cannot place a value into `m_buffer`.

Member Function `put` and the Producer Thread

11 You'll synchronize access to `SynchronizedBuffer`'s shared mutable data using the class's **`condition_variable`**, **`mutex`** and a C++11 **`std::unique_lock`**, which among its capabilities can lock a **`mutex`** that's used in conjunction with a **`condition_variable`**. When the producer thread invokes **`put`**, it must first acquire `m_mutex`'s lock to ensure that it has exclusive access to `SynchronizedBuffer`'s shared mutable data. You **acquire a `mutex`'s lock** by creating a **lock object** and initializing it with the **`mutex`** (line 20). If the **`mutex`'s lock** is not available, the thread creating the lock object is **blocked** and must **wait** until it can acquire the lock. Once the thread acquires the lock, the rest of that block is said to be guarded by the **`mutex`**.

Err As you'll see momentarily, a **`unique_lock`** can **release a `mutex`'s lock** when it's not a given thread's turn to perform its task, then can **reacquire the lock later**. This capability is important. Holding a lock on the `SynchronizedBuffer`'s **`mutex`** when a thread cannot perform its task could cause **deadlock**.

Acquiring the Lock and Checking Whether It's the Producer's Turn

When the `m_mutex`'s lock is available, line 20 **acquires the lock**. Then, line 22 checks whether `m_occupied` is `true`. If so, `m_buffer` is full, and the **producer thread must wait** until `m_buffer` is empty, so lines 23–25 output a message indicating that

55. C++ Core Guidelines, "CP.50: Define a `mutex` Together with the Data It Guards." Accessed February 7, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-mutex.

- the producer thread is trying to write a value,

- `m_buffer` is full and
- the producer thread is waiting until there's space.

Waiting on the Condition Variable

Condition variables can be used to make a thread **wait** while a condition is not satisfied then to **notify a waiting thread** to proceed when a condition is satisfied:

If the producer thread obtains the **`mutex`'s** lock, but the buffer is full, the thread calls **`condition_variable`'s** **`wait`** function (line 30), passing the **`unique_lock`** (`dataLock`) as the first argument. This causes `dataLock` to **release the `mutex`'s lock**, places the producer thread in **`condition_variable` `m_cv`'s waiting state** and removes the thread from contention for a processor. **This is important because the producer thread cannot currently perform its task. So, the consumer should be allowed to access the `SynchronizedBuffer` to allow the condition (`m_occupied`) to change**. Now the consumer can attempt to **acquire the lock** on `m_mutex`.

When the consumer thread executing the critical section in function **`get`** satisfies the condition on which the producer thread may be waiting—that is, the consumer reads the buffer's value, so **the buffer is empty**—the consumer thread calls **`notify_one`** on **`m_cv`** (line 75). This allows the producer thread waiting on that condition to transition to the **ready state**.[56] When the operating system moves the producer thread to the **running state**, it can attempt to **reacquire the lock**.[57]

56. From a December 6, 2021, email correspondence with Anthony Williams: "Condition variables do not specify which thread is woken when `notify_one` is called. It is valid for *all* the waiting threads to be woken. ... In particular, it is often most efficient from an OS perspective to allow the thread that has been waiting the **shortest time** to be woken ..., as its data might still be in cache."

57. In some applications, when a thread reacquires the lock, it still might not be able to perform its task—in which case, it will reenter the condition variable's waiting state and implicitly release the lock.

Once the producer is notified and **implicitly reacquires `m_mutex`'s lock**, **`put`** continues executing with the next statement after the **`wait`** call.

Spurious Wakeup

Occasionally, the system could move a waiting thread back into the **ready state** before that thread can perform its task—this is known as a **spurious wakeup**.[58, 59] The **`wait`** function's second argument in line 30 is a no-argument lambda that checks whether it's the producer thread's turn to access `m_buffer`. When the producer thread gets the processor, it first reacquires the `m_mutex`'s lock, then calls this lambda. If the lambda returns `false`, the thread releases the lock and **returns to `condition_variable` `m_cv`'s waiting state**; otherwise, the producer thread continues executing.

58. Marius Bancila, *Modern C++ Programming Cookbook, 2/e*, p. 422. Birmingham, UK: Packt Publishing. 2020.

59. "Spurious Wakeup." Accessed February 7, 2022. https://en.wikipedia.org/wiki/Spurious_wakeup.

Updating the `SynchronizedBuffer`'s State

Line 34 assigns the produced `value` to `m_buffer`. Line 35 sets `m_occupied` to `true` to indicate that `m_buffer` now contains a value (i.e., a consumer can read the value, but a producer cannot yet put another value there). Lines 37–39 indicate that the producer is writing a value into the `m_buffer`.

Releasing a Lock

At this point, execution reaches line 40—the end of the block that defines `put`'s **critical section**. When a thread finishes using a shared object that's managed with `unique_lock`, the thread **must release the lock** by calling the lock's `unlock` function implicitly or explicitly. Lock objects use **RAII** (discussed in Section 11.5). If you do not explicitly call `unlock`, class `unique_lock`'s destructor will implicitly call it when the lock goes out of scope at the end of the **critical section**. If the consumer thread was previously **blocked** while attempting to enter `get`'s critical section, which is guarded by the same `mutex`, the consumer thread now can acquire the lock to proceed.

Notifying the Consumer Thread to Continue

Perf Now that **the buffer is full**, the producer thread calls `m_cv`'s `notify_one` function (line 43) to indicate that the buffer's state has changed. If the consumer is **waiting** on this condition variable, it enters the **ready state** and becomes eligible to **reacquire the lock**. Function `notify_one` returns immediately, then `put` returns to its caller (i.e., the producer thread). For performance, you should unlock the `unique_lock` before calling `notify_one` (or `notify_all`) to ensure that the notified waiting thread does not need to wait for the `mutex` lock to become available.[60]

Member Function `get` and the Consumer Thread

Functions `get` and `put` are implemented similarly. When the consumer thread invokes `get`, line 53 creates a `unique_lock` and attempts to acquire `m_mutex`'s lock. If the lock is not available (e.g., the producer has not yet released it), the **consumer thread is blocked until the lock becomes available**. If it is available, line 53 acquires it, so the **consumer thread now owns the lock**. Next, line 55 checks whether `m_occupied` is `false`. If so, `m_buffer` is empty, so lines 56–58 indicate that

- the consumer thread is trying to read a value,
- the buffer is empty and
- the consumer thread is waiting.

Waiting on the Condition Variable

Line 63 invokes the **`m_cv`'s `wait` function** to place the consumer thread in that `condition_variable`'s **waiting state**. Again, `wait` causes `dataLock` to **release `m_mutex`'s lock**, so **the producer thread can attempt to acquire it** and do its work.

The consumer thread remains in the **waiting state** until it's **notified** by the producer thread to proceed. At that point, the **consumer thread** returns to the **ready state**. When the operating system moves the thread to the **running state**, the **consumer thread** attempts to **implicitly reacquire `m_mutex`'s lock**. If it's available, the **consumer thread** reacquires it and `get` continues executing with the next statement after `wait`. The second argument to `m_cv`'s `wait` function is a lambda that checks whether it's the consumer thread's turn. If the consumer thread gets a processor and this lambda returns `false`, the thread will return to `m_cv`'s **waiting state**.

60. Bancila, *Modern C++ Programming Cookbook*, p. 420.

Updating the `SynchronizedBuffer`'s State and Notifying the Producer to Continue

Line 66 stores the `value` that will be returned to the calling thread, line 67 sets `m_occupied` to `false` to indicate that `m_buffer` is now empty (i.e., the producer can produce), and lines 69–71 indicate the consumer is reading a value. At this point, `dataLock` goes out of scope and its **destructor releases `m_mutex`'s lock**. Next, line 75 invokes `m_cv`'s `notify_one` function. If a producer thread is

in `m_cv`'s **waiting state**, it enters the **ready state** and becomes eligible to **reacquire `m_mutex`'s lock**. Function `notify_one` returns immediately, then `get` returns `value` to its caller.

Pairing Waits and Notifications

When concurrent threads manipulate a shared object using locks on a given `mutex`, ensure that if one thread calls function `wait` to enter a `condition_variable`'s **waiting state**, a separate thread eventually will call `condition_variable` function `notify_one` to transition the waiting thread back to the **ready state**.

17.6.2 Testing `SynchronizedBuffer`

11 17 Figure 17.8 is similar to Fig. 17.6. Lines 19–24 define the lambda `getSleepTime`, which uses a `std::mutex` (line 16) and a C++11 `std::lock_guard` (line 21) to ensure synchronized access to a random-number generator that we'll share among this example's threads. We do this only for demonstration purposes in this example to show using a `lock_guard` to protect a shared resource that does not require a `condition_variable`. When you construct a `lock_guard`, it **acquires its `mutex` argument's lock** if it's available; otherwise, the thread creating the `lock_guard` is **blocked** until it can acquire the lock. Unlike `unique_lock`, which can release and reacquire a `mutex`'s lock—capabilities we need to use `condition_variable`s—a `lock_guard` **owns a `mutex`'s lock until the `lock_guard` goes out of scope**. At that point, its **destructor releases the `mutex`'s lock.**[61] Thus, only one thread at a time can execute `getSleepTime`'s block. For threads that need to access resources guarded by separate `mutex`es, C++ also provides `std::scoped_lock` (C++17), which acquires locks on several `mutex`es at once and releases them all when the `scoped_lock` goes out of scope at the end of its enclosing block.[62]

61. "`std::lock_guard`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/lock_guard.

62. "`std::scoped_lock`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/scoped_lock.

Line 27 creates the `SynchronizedBuffer` that we share between the concurrent producer and consumer threads. Lines 30–44 and 47–61 define the `produce` and `consume` lambdas that will be called by the producer and consumer `jthread`s (lines 67–68). Each lambda captures `buffer` and `getSleepTime` by reference. Lines 63–65 display the output's column heads. When function `main` ends, the `jthread`s go out of scope, automatically joining the threads, enabling them to complete execution before the program terminates.

```
1  // Fig. 17.8: SharedBufferTest.cpp
2  // Concurrent threads correctly manipulating a synchronized buffer.
3  #include <chrono>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <mutex>
7  #include <random>
8  #include <thread>
9  #include "SynchronizedBuffer.h"
10
11 int main() {
12    // set up random-number generation
13    std::random_device rd;
14    std::default_random_engine engine{rd()};
```

```
15      std::uniform_int_distribution ints{0, 3000};
16      std::mutex intsMutex;
17
18      // lambda for synchronized random sleep time generation
19      auto getSleepTime{
20         [&]() {
21            std::lock_guard lock{intsMutex};
22            return std::chrono::milliseconds{ints(engine)};
23         }
24      };
25
26      // create SynchronizedBuffer to store ints
27      SynchronizedBuffer buffer;
28
29      // lambda expression that produces the values 1-10 and sums them
30      auto produce{
31         [&buffer, &getSleepTime]() {
32            int sum{0};
33
34            for (int count{1}; count <= 10; ++count) {
35               // get random sleep time then sleep
36               std::this_thread::sleep_for(getSleepTime());
37
38               buffer.put(count); // set value in buffer
39               sum += count; // add count to sum of values produced
40            }
41
42            std::cout << "Producer done producing\nTerminating Producer\n";
43         }
44      };
45
46      // lambda expression that consumes the values 1-10 and sums them
47      auto consume{
48         [&buffer, &getSleepTime]() {
49            int sum{0};
50
51            for (int count{1}; count <= 10; ++count) {
52               // get random sleep time then sleep
53               std::this_thread::sleep_for(getSleepTime());
54
55               sum += buffer.get(); // get buffer value and add to sum
56            }
57
58            std::cout << fmt::format("\n{} {}\n{}\n",
59               "Consumer read values totaling", sum, "Terminating Consumer");
60         }
61      };
62
63      std::cout << fmt::format("{:<40}{}\t\t{}\n{:<40}{}\t\t{}\n\n",
64         "Operation", "Buffer", "Occupied",
65         "---------", "------", "--------");
66
67      std::jthread producer{produce}; // start producer thread
68      std::jthread consumer{consume}; // start consumer thread
```

```
69   }
```

```
Operation                          Buffer         Occupied
---------                          ------         --------

Consumer tries to read.
Buffer empty. Consumer waits.      -1             false

Producer writes 1                  1              true

Consumer reads 1                   1              false
-----------------------------------------------------------------
Producer writes 2                  2              true

Producer tries to write.
Buffer full. Producer waits.       2              true

Consumer reads 2                   2              false
-----------------------------------------------------------------
Producer writes 3                  3              true

Consumer reads 3                   3              false
-----------------------------------------------------------------
Consumer tries to read.
Buffer empty. Consumer waits.      3              false

Producer writes 4                  4              true

Consumer reads 4                   4              false
-----------------------------------------------------------------
Producer writes 5                  5              true

Consumer reads 5                   5              false
-----------------------------------------------------------------
Producer writes 6                  6              true

Producer tries to write.
Buffer full. Producer waits.       6              true

Consumer reads 6                   6              false
-----------------------------------------------------------------
Producer writes 7                  7              true

Producer tries to write.
Buffer full. Producer waits.       7              true

Consumer reads 7                   7              false
-----------------------------------------------------------------
Producer writes 8                  8              true

Producer tries to write.
Buffer full. Producer waits.       8              true

Consumer reads 8                   8              false
-----------------------------------------------------------------
Producer writes 9                  9              true
```

```
Consumer reads 9                    9              false
------------------------------------------------------------
Producer writes 10                  10             true

Producer done producing
Terminating Producer
Consumer reads 10                   10             false
------------------------------------------------------------

Consumer read values totaling 55
Terminating Consumer
```

Fig. 17.8 Concurrent threads correctly manipulating a synchronized buffer.

Analyzing the Output

Study the output in Fig. 17.8 and observe that:

- Every integer produced is consumed exactly once—**no values are lost**, and **no values are consumed more than once**.
- The synchronization ensures the producer produces a value only when the buffer is empty, and the consumer consumes only when the buffer is full.
- The producer always produces a value before the consumer is allowed to consume a value for the first time.
- The consumer waits if the producer has not produced since the consumer last consumed.
- The producer waits if the consumer has not yet consumed the value that the producer most recently produced.

Execute this program several times to confirm that **every integer produced is consumed exactly once**. In the sample output, we applied bold to the lines indicating when the producer and consumer must wait to perform their respective tasks.

Note Regarding Output Statements in Our Synchronization Examples

CG Perf In addition to performing the actual operations that manipulate the `SynchronizedBuffer`, our **synchronized `put` and `get` functions** print messages to the console indicating the threads' progress as they execute these functions. We do this so the messages will print in the correct order, showing that `put` and `get` are correctly synchronized. We continue to output messages from critical sections in later examples for demonstration purposes only. **The C++ Core Guidelines say to minimize the duration of critical sections**[63]**—**that is, the amount of time an object is "locked." **Avoid performing I/O, lengthy calculations and operations that do not require synchronization while holding a lock.**

63. C++ Core Guidelines, "CP.43: Minimize Time Spent in a Critical Section." Accessed February 7, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-time.

SE Also, for demonstration purposes, lines 36 and 53 call `sleep_for` to emphasize the unpredictability of thread scheduling. Though we are not doing so here, it's important to note that **a thread should never sleep while holding a lock in a real application.**

Perf The program in Section 17.6 used thread synchronization to guarantee that two concurrent threads correctly manipulated data in a shared buffer. However, the application may not perform optimally. If the threads operate at different speeds, one will spend more (or most) of its time waiting. For example:

- If the producer thread produces values faster than the consumer can consume them, the producer thread **waits** because there are no empty locations for writing.
- If the consumer consumes values faster than the producer produces them, the consumer **waits** until the producer places the next value in the shared buffer.

Even threads that operate at approximately the same relative speeds can occasionally become "out of sync" over a period of time, causing one of them to **wait** for the other.

Perf **We cannot predict the relative speeds of asynchronous concurrent threads.** Interactions with the operating system, the network, the user and other components can cause threads to operate at different and ever-changing speeds. When this happens, threads wait. **When threads wait excessively, programs can become less efficient, interactive programs can become less responsive, and applications can suffer potentially long delays.**

Circular Buffers

Using a **circular buffer**, we can minimize **waiting** among concurrent threads that share resources and operate at the same average speeds. Such a buffer provides a fixed number of cells into which the producer can write values and from which the consumer can read those values. Internally, a circular buffer manages the producer's writes into the buffer and the consumer's reads from the buffer elements in order, beginning at the first cell and moving toward the last. When the circular buffer reaches its last element, it "wraps around" to the first element and continues from there—thus the name "circular."

An example of the producer–consumer relationship that uses a circular buffer is the video streaming we discussed in Section 17.1. With a circular buffer:

- **If the producer temporarily operates faster than the consumer, the producer can write additional values into the extra buffer cells, if any are available**. This enables the producer to keep busy even though the consumer is not ready to retrieve the current value being produced.
- **If the consumer temporarily operates faster than the producer, the consumer can read additional values (if there are any) from the buffer**. This enables the consumer to keep busy even though the producer is not ready to produce additional values.

Even with a **circular buffer**, a producer thread could fill the buffer, forcing the producer to **wait** until a consumer consumed a value to free an element in the buffer. Similarly, if the buffer is empty, a consumer must **wait** until the producer produces another value.

Even a **circular buffer** is inappropriate if the producer and the consumer operate consistently at significantly different average speeds:

- If the **consumer always executes faster**, a buffer containing one location (or a small number of locations) is enough.
- If the **producer executes faster on average** and the program does not ask the producer to wait, only a buffer with an "infinite" number of locations would absorb the extra production.

SE If they **execute at about the same average speed**, a circular buffer helps to smooth the effects of occasional speeding up or slowing down in either thread's execution and reduces waiting times, improving performance.

Perf **The key to a circular buffer is optimizing its size to minimize thread wait times while not wasting memory**. If we determine that the producer often produces as many as three more values than the consumer can consume, we can provide a buffer of at least three cells to handle the extra production. Making the buffer too small would cause threads to wait longer.

Implementing a Circular Buffer

Figures 17.9—17.10 demonstrate concurrent producer and consumer threads accessing a **circular buffer with synchronization** (Fig. 17.9). We implement the circular buffer as an `array` of three `int` elements. **The consumer consumes a value only when the `array` is not empty, and the producer produces a value only when the `array` is not full.** Again, the output statements used in this class's critical sections are for demonstration purposes only.

```
1   // Fig. 17.9: CircularBuffer.h
2   // Synchronizing access to a shared three-element circular buffer.
3   #pragma once
4   #include <array>
5   #include <condition_variable>
6   #include <fmt/format.h>
7   #include <mutex>
8   #include <iostream>
9   #include <string>
10  #include <string_view>
11
12  using namespace std::string_literals;
13
14  class CircularBuffer {
15  public:
16     // place value into m_buffer
17     void put(int value) {
18        // critical section that requires a lock to modify shared data
19        {
20           // lock on m_mutex to be able to write to m_buffer
21           std::unique_lock dataLock{m_mutex};
22
23           // if no empty locations, wait on condition variable m_cv
24           if (m_occupiedCells == m_buffer.size()) {
25              std::cout << "Buffer is full. Producer waits.\n\n";
26
27              // wait on the condition variable; the lambda argument
28              // ensures that if the thread gets the processor before
29              // the m_buffer has open cells, the thread continues waiting
30              m_cv.wait(dataLock,
```

```
                [&] {return m_occupiedCells < m_buffer.size();});
         }

         m_buffer[m_writeIndex] = value; // write to m_buffer
         ++m_occupiedCells; // one more m_buffer cell is occupied
         m_writeIndex = (m_writeIndex + 1) % m_buffer.size();
         displayState(fmt::format("Producer writes {}", value));
      } // dataLock's destructor releases the lock on m_mutex here

      m_cv.notify_one(); // notify threads waiting to read from m_buffer
   }

   // return value from m_buffer
   int get() {
      int readValue; // will temporarily hold the next value read

      // critical section that requires a lock to modify shared data
      {
         // lock on m_mutex to be able to write to m_buffer
         std::unique_lock dataLock{m_mutex};

         // if no data to read, place thread in waiting state
         if (m_occupiedCells == 0) {
            std::cout << "Buffer is empty. Consumer waits.\n\n";

            // wait on the condition variable; the lambda argument
            // ensures that if the thread gets the processor before
            // there is data in the m_buffer, the thread continues waiting
            m_cv.wait(dataLock, [&]() {return m_occupiedCells > 0;});
         }

         readValue = m_buffer[m_readIndex]; // read value from m_buffer
         m_readIndex = (m_readIndex + 1) % m_buffer.size();
         --m_occupiedCells;// one fewer m_buffer cells is occupied
         displayState(fmt::format("Consumer reads {}", readValue));
      } // dataLock's destructor releases the lock on m_mutex here

      m_cv.notify_one();// notify threads waiting to write to m_buffer
      return readValue;
   }

   // display current operation and m_buffer state
   void displayState(std::string_view operation) const {
      std::string s;

      // add operation argument and number of occupied m_buffer cells
      s += fmt::format("{} (buffer cells occupied: {})\n{:<15}",
         operation, m_occupiedCells, "buffer cells:");

      // add values in m_buffer
      for (int value : m_buffer) {
         s += fmt::format(" {:2d} ", value);
      }

      s += fmt::format("\n{:<15}", ""); // padding
```

```
86
87          // add underlines
88          for (int i{0}; i < m_buffer.size(); ++i) {
89             s += "---- "s;
90          }
91
92          s += fmt::format("\n{:<15}", ""); // padding
93
94          for (int i{0}; i < m_buffer.size(); ++i) {
95             s += fmt::format(" {}{} ",
96                (i == m_writeIndex ? 'W' : ' '),
97                (i == m_readIndex ? 'R' : ' '));
98          }
99
100         s += "\n\n";
101         std::cout << s;// display the state string
102      }
103   private:
104      std::condition_variable m_cv;
105      std::mutex m_mutex;
106
107      std::array<int, 3> m_buffer{-1, -1, -1}; // shared m_buffer
108      int m_occupiedCells{0}; // count number of buffers used
109      int m_writeIndex{0};// index of next element to write to
110      int m_readIndex{0};// index of next element to read
111   };
```

Fig. 17.9 Synchronizing access to a shared three-element circular buffer.

Data Members

Lines 104–110 define the class's data members:

- Lines 104–105 define the `std::condition_variable` `m_cv` and the `std::mutex` `m_mutex` for **synchronizing access** to `CircularBuffer`'s other data members.
- Line 107 creates and initializes the three-element `int array` `m_buffer`, representing the **circular buffer**.
- Variable `m_occupiedCells` (line 108) counts the number of `m_buffer` elements that contain readable data. When `m_occupiedCells` is `0`, **the circular buffer is empty, and the consumer must wait**. When `m_occupiedCells` is `3` (the buffer's size), the **circular buffer is full, and the producer must wait**.
- Variable `m_writeIndex` (line 109) indicates the next location in which a value can be placed by a producer.
- Variable `m_readIndex` (line 110) indicates the position from which the next value can be read by a consumer.

Data members `m_buffer`, `m_occupiedCells`, `m_writeIndex` and `m_readIndex` are all part of the class's **shared mutable data**, so **access to these variables must be synchronized** to ensure that a `CircularBuffer` is **thread-safe**.

`CircularBuffer` Function put

The `put` function (lines 17–41) performs the same tasks as in Fig. 17.7, with a few modifications. Lines 24–32 check whether the `CircularBuffer` is full. If so, the **producer must wait**, so line 25 indicates that the `Producer` is **waiting** to perform its task. Then, lines 30–31 invoke `m_cv`'s `wait` function, causing the

producer thread to **release `m_mutex`'s lock** and **wait** until there's space in the buffer to write a new value.

When execution continues at line 34, the producer places `value` in the **circular buffer** at location `m_writeIndex`. Line 35 increments `occupiedCells`, because the buffer contains a value the consumer can read. Then line 36 updates `m_writeIndex` for the producer's next `put` call. **This line is the key to the buffer's circularity**. When `writeIndex` increments **past the end of the buffer**, this line sets it back to `0`. Next, line 37 calls `displayState` (lines 73–102) to display the value the producer wrote, the `occupiedCells` count, the cells' contents and the current `m_writeIndex` and `m_readIndex`. Reaching the end of the block at line 38 releases **`m_mutex`'s lock**. Line 40 calls **`notify_one`** on `m_cv` to transition a **waiting** thread to the **ready state** so that a **waiting consumer thread** (if there is one) can now try again to read a value from the buffer.

`CircularBuffer` Function `get`

The **`get`** function (lines 44–70) performs the same tasks as in Fig. 17.7, with a few minor modifications. Lines 53–60 (Fig. 17.9) determine whether all the buffer cells are **empty**, in which case the **consumer must wait**. If so, line 54 updates the output to indicate that the **consumer is waiting** to perform its task. Then line 59 invokes **`m_cv`'s `wait`** function, causing the consumer thread to **release `m_mutex`'s lock** and **wait** until data is available to read.

When execution continues at line 62, the local variable `readValue` is assigned the value at `m_buffer` location `m_readIndex`. Then line 63 updates `m_readIndex` for the next call to `CircularBuffer` function **`get`**. **This line and line 36 implement the buffer's circularity**. Line 64 decrements `m_occupiedCells`, because there's now an open buffer position in which the producer can place a value. Line 65 updates the output with the value being consumed, the `occupiedCells` count, the cells' contents and the current `m_writeIndex` and `m_readIndex`. Reaching line 66 releases **`m_mutex`'s lock**. Line 68 calls `m_cv`'s **`notify_one`** function to allow a waiting producer thread to attempt to write again. Then line 69 returns the consumed value to the caller.

`CircularBuffer` Function `displayState`

Function `displayState` (lines 73–102) builds then outputs a `string` containing the application's state. Lines 81–83 add the buffer cells' values to the string, using a `":2d"` format specifier to format the contents of each cell with a leading space if it's a single digit. Lines 94–98 add to the `string` the current `m_writeIndex` and `m_readIndex` with the letters `W` and `R`, respectively. **Note that we call this function only from the critical sections to ensure thread-safety. Again, we should not do I/O in critical sections—we do this simply to produce useful outputs for demonstration purposes.**

Testing Class `CircularBuffer`

Figure 17.10 contains the `main` function that launches the application. Line 13 creates a `CircularBuffer` object named `buffer`, and line 14 displays `buffer`'s initial state. Lines 17–37 and 40–60 create the `produce` and `consume` lambdas that the concurrent producer and consumer threads will execute. Lines 62–63 create the **`std::jthread`s** that call the `produce` and `consume` lambdas. When `main` ends, the **`std::jthread`s** go out of scope, automatically joining the threads.

```
1  // Fig. 17.10: SharedBufferTest.cpp
2  // Concurrent threads manipulating a synchronized circular buffer.
3  #include <chrono>
4  #include <fmt/format.h>
5  #include <iostream>
```

```
#include <mutex>
#include <random>
#include <thread>
#include "CircularBuffer.h"

int main() {
   // create CircularBuffer to store ints and display initial state
   CircularBuffer buffer;
   buffer.displayState("Initial State");

   // lambda expression that produces the values 1-10 and sums them
   auto produce{
      [&buffer]() {
         // set up random-number generation
         std::random_device rd;
         std::default_random_engine engine{rd()};
         std::uniform_int_distribution ints{0, 3000};

         int sum{0};


         for (int count{1}; count <= 10; ++count) {
            // get random sleep time then sleep
            std::chrono::milliseconds sleepTime{ints(engine)};
            std::this_thread::sleep_for(sleepTime);

            buffer.put(count);// set value in buffer
            sum += count; // add count to sum of values produced
         }

         std::cout << "Producer done producing\nTerminating Producer\n\n";
      }
   };

   // lambda expression that consumes the values 1-10 and sums them
   auto consume{
      [&buffer]() {
         // set up random-number generation
         std::random_device rd;
         std::default_random_engine engine{rd()};
         std::uniform_int_distribution ints{0, 3000};

         int sum{0};

         for (int count{1}; count <= 10; ++count) {
            // get random sleep time then sleep
            std::chrono::milliseconds sleepTime{ints(engine)};
            std::this_thread::sleep_for(sleepTime);

            sum += buffer.get(); // get buffer value and add to sum
         }

         std::cout << fmt::format("{} {}\n{}\n\n",
            "Consumer read values totaling", sum, "Terminating Consumer");
      }
```

```
60      };
61
62      std::jthread producer{produce}; // start producer thread
63      std::jthread consumer{consume}; // start consumer thread
64   }
```

```
Initial State (buffer cells occupied: 0)
buffer cells:   -1   -1   -1
               ---- ---- ----
                WR

Buffer is empty. Consumer waits.

Producer writes 1 (buffer cells occupied: 1)
buffer cells:    1   -1   -1
               ---- ---- ----
                 R    W

Consumer reads 1 (buffer cells occupied: 0)
buffer cells:    1   -1   -1
               ---- ---- ----
                     WR

Buffer is empty. Consumer waits.

Producer writes 2 (buffer cells occupied: 1)
buffer cells:    1    2   -1
               ---- ---- ----
                      R    W

Consumer reads 2 (buffer cells occupied: 0)
buffer cells:    1    2   -1
               ---- ---- ----
                          WR

Buffer is empty. Consumer waits.

Producer writes 3 (buffer cells occupied: 1)
buffer cells:    1    2    3
               ---- ---- ----
                W          R

Consumer reads 3 (buffer cells occupied: 0)
buffer cells:    1   2    3
               ---- ---- ----
                WR

Producer writes 4 (buffer cells occupied: 1)
buffer cells:    4    2    3
                ---- ---- ----
                  R   W

Producer writes 5 (buffer cells occupied: 2)
buffer cells:    4    5    3
               ---- ---- ----
                 R         W
```

```
Consumer reads 4 (buffer cells occupied: 1)
buffer cells:    4    5    3
                ---- ---- ----
                      R    W

Consumer reads 5 (buffer cells occupied: 0)
buffer cells:    4    5    3
                ---- ---- ----
                          WR

Producer writes 6 (buffer cells occupied: 1)
buffer cells:    4    5    6
                ---- ---- ----
                  W         R

Producer writes 7 (buffer cells occupied: 2)
buffer cells:    7    5    6
                ---- ---- ----
                      W    R

Producer writes 8 (buffer cells occupied: 3)
buffer cells:    7    8    6
                ---- ---- ----
                          WR

Buffer is full. Producer waits.

Consumer reads 6 (buffer cells occupied: 2)
buffer cells:    7    8    6
                ---- ---- ----
                  R         W

Producer writes 9 (buffer cells occupied: 3)
buffer cells:    7    8    9
                ---- ---- ----
                 WR

Buffer is full. Producer waits.

Consumer reads 7 (buffer cells occupied: 2)
buffer cells:    7    8    9
                ---- ---- ----
                  W    R

Producer writes 10 (buffer cells occupied: 3)
buffer cells:   10    8    9
                ---- ---- ----
                     WR

Producer done producing
Terminating Producer

Consumer reads 8 (buffer cells occupied: 2)
buffer cells:   10    8    9
```

```
                ---- ---- ----
                     W    R

Consumer reads 9 (buffer cells occupied: 1)
buffer cells:   10    8    9
                ---- ---- ----
                 R    W

Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10    8    9
                ---- ---- ----
                     WR

Consumer read values totaling 55
Terminating Consumer
```

Fig. 17.10 Concurrent threads manipulating a synchronized circular buffer.

Analyzing the Output

When the producer writes a value or the consumer reads a value, the program outputs a message indicating the action performed, `m_buffer`'s contents, and the `m_writeIndex` (`"W"`) and `m_readIndex` (`"R"`) locations. In the sample output, **the consumer immediately waits because it tries to consume before the producer executes**. Next, the producer writes `1`. The buffer then contains `1` in the first cell and `-1` (the default value we use for output purposes) in the other two cells. The write index is now at the second cell, while the read index is still at the first cell. Next, the consumer reads `1`. The buffer contains the same values, but the read index is now at the second cell. **The consumer then tries to read again, but the buffer is empty, and the consumer must wait**. The consumer also **waits** after reading `2`. Later in the output, **the producer fills the buffer twice and subsequently waits each time**.

17.8 Readers and Writers

Our producer–consumer examples used a single producer and a single consumer—common in many applications with threads that share mutable data. Some systems require multiple consumer threads (called readers) that read data and multiple producer threads (called writers) that write it. This is known as the **readers and writers problem**.

For example, there may be many more readers than writers in an airline reservation system. Many inquiries will be made against the database of available flight information before a customer actually purchases a particular seat on a particular flight. The key observation is that if you have multiple readers, **they can read simultaneously** without thread-safety issues because they do not modify the data. A writer still requires exclusive access to the critical section that modifies the data—there can be no other writers and no readers.

The C++ features that support multiple readers and writers are `std::shared_mutex`, `std::shared_lock` and `std::condition_variable_any`. A `std::shared_mutex`[64] (from the `<mutex>` header) **allows one writer or multiple readers to own its lock**:

- If the lock is available, a writer acquires it via a `lock_guard` or a `unique_lock`. Otherwise, the writer is **blocked** and must wait for the lock to become available. **While a writer holds a `shared_mutex`'s lock, no other threads may acquire it.**

- If the lock is available, a reader acquires it via a `std::shared_lock`[65] (from the `<mutex>` header); otherwise, the reader is blocked and must wait for the lock to become available. While a reader holds a `shared_mutex`'s lock, only other readers may acquire the lock.

64. "`std::shared_mutex`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/shared_mutex.

65. "`std::shared_lock`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/shared_lock.

As in our synchronized producer–consumer examples, we must ensure that readers and writers perform their tasks only when it's their turn:

- If a reader is reading when a writer arrives, the writer must **wait** for the lock to become available. Also, any additional readers that subsequently arrive must **wait** until the currently waiting writer executes and notifies them that it has finished writing. The waiting writer would be **indefinitely postponed** if a stream of arriving readers were allowed to read.
- If a writer is writing when a reader arrives, the reader must **wait** for the lock to become available. Also, any additional writers that subsequently arrive must **wait** until the currently waiting readers read. The waiting readers would be **indefinitely postponed** if a stream of arriving writers were allowed to write.

Once again, we can use condition variables to manage these conditions—one for the readers and a separate one for the writers.

Readers use **`shared_lock`**s, but class **`condition_variable`** requires **`unique_lock`**s. So, for readers, use a **`std::condition_variable_any`**[66] object to manage waiting readers. Class **`condition_variable_any`** works like **`condition_variable`** but supports other lock types. When a writer finishes writing, it would **notify all currently waiting readers** that it's time to read by calling the **`condition_variable_any`** object's **`notify_all`** function. All reader threads waiting for that condition would then transition to the **ready state** and become eligible to reacquire the lock.

66. "`std::condition_variable`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/condition_variable_any.

Writer thread(s) would still use **`unique_lock`** for exclusive access to the shared mutable data, so we can manage waiting writers with a **`condition_variable`**. When all active readers finish reading, the program would call the **`condition_variable`** object's **`notify_all`** function. All **the waiting writer threads would move to the ready state** and become eligible to reacquire the lock. However, **only one writer would reacquire the lock** and proceed.

17.9 Cooperatively Canceling **`jthread`**s

20 **When a multithreaded application needs to terminate, it's good practice to shut down threads that are still performing tasks, so they can release resources they're using.**[67] For example, they might need to close files, database connections and network connections. Before C++20, there was no standard mechanism for threads to cooperate with one another to terminate gracefully. This is another defect of **`thread`**.

67. Anthony Williams, "Concurrency in C++20 and Beyond," October 16, 2019. Accessed February 7, 2022. https://www.youtube.com/watch?v=jozHW_B3D4U.

20 C++20 added **cooperative cancellation** to fix this problem, enabling programs to notify threads when it's time for them to terminate. The task executing in a thread can watch for such notifications, then complete critical work, release resources and terminate.

Figure 17.11 demonstrates **cooperative cancellation between threads** using features from the `<stop_token>` **header**. As you'll see, `jthread` integrates these features.

```
// Fig. 17.11: CooperativeCancelation.cpp
// Using a std::jthread's built-in stop_source
// to request that the std::jthread stop executing.
#include <chrono>
#include <fmt/format.h>
#include <iostream>
#include <mutex>
#include <random>

#include <sstream>
#include <string>
#include <string_view>
#include <thread>

// get current thread's ID as a string
std::string id() {
   std::ostringstream out;
   out << std::this_thread::get_id();
   return out.str();
}

int main() {
   // each printTask iterates until a stop is requested by another thread
   auto printTask{
      [&](std::stop_token token, std::string name) {
         // set up random-number generation
         std::random_device rd;
         std::default_random_engine engine{rd()};
         std::uniform_int_distribution ints{500, 1000};

         // register a function to call when a stop is requested
         std::stop_callback callback(token, [&]() {
            std::cout << fmt::format(
               "{} told to stop by thread with id {}\n", name, id());
         });

         while (!token.stop_requested()) { // run until stop requested
            auto sleepTime{std::chrono::milliseconds{ints(engine)}};

            std::cout << fmt::format(
               "{} (id: {}) going to sleep for {} ms\n",
               name, id(), sleepTime.count());

            // put thread to sleep for sleepTime milliseconds
            std::this_thread::sleep_for(sleepTime);

            // show that task woke up
            std::cout << fmt::format("{} working.\n", name);
         }

         std::cout << fmt::format("{} terminating.\n", name);
```

```
51          }
52       };
53
54       std::cout << fmt::format("MAIN (id: {}) STARTING TASKS\n", id());
55
56       // create two jthreads that each call printTask with a string argument
57       std::jthread task1{printTask, "Task 1"};
58       std::jthread task2{printTask, "Task 2"};

60       // put main thread to sleep for 2 seconds
61       std::cout << "\nMAIN GOING TO SLEEP FOR 2 SECONDS\n\n";
62       std::this_thread::sleep_for(std::chrono::seconds{2});
63
64       std::cout << fmt::format("\nMAIN (id: {}) ENDS\n\n", id());
65    }
```

```
MAIN (id: 16352) STARTING TASKS

MAIN GOING TO SLEEP FOR 2 SECONDS

Task 1 (id: 14048) going to sleep for 708 ms
Task 2 (id: 10504) going to sleep for 995 ms
Task 1 working.
Task 1 (id: 14048) going to sleep for 940 ms
Task 2 working.
Task 2 (id: 10504) going to sleep for 926 ms
Task 1 working.
Task 1 (id: 14048) going to sleep for 875 ms
Task 2 working.
Task 2 (id: 10504) going to sleep for 519 ms

MAIN (id: 16352) ENDS

Task 2 told to stop by thread with id 16352
Task 2 working.
Task 2 terminating.
Task 1 told to stop by thread with id 16352
Task 1 working.
Task 1 terminating.
```

Fig. 17.11 Using a `std::jthread`'s built-in `stop_source` to request that the `std::jthread` stop executing.

Each **`jthread`** internally maintains a **`std::stop_source`**, which has an associated **`std::stop_token`**. A **`jthread`**'s **task function** optionally can receive this token as its first parameter. The task function then can periodically call the token's **`stop_requested` member function** to determine whether the task should stop executing. When

- another thread calls the **`jthread`'s `request_stop` member function**, or
- the **`jthread`'s destructor** calls **`request_stop`** as the **`jthread`** goes out of scope,

the **`stop_source`** notifies its associated **`stop_token`** that a **stop has been requested**. Subsequent calls to the **`stop_token`'s `stop_requested` member function** will return `true`. The task can then gracefully terminate its execution. If the task function never calls **`stop_requested`**, the corresponding **`jthread`** continues executing—hence, the term **cooperative cancellation**.

In this example, we'll let the **`jthread`**'s destructor call the **`request_stop`** member function, enabling the program to terminate soon after `main` completes. We create two **`jthread`s** (lines 57–58) that call the `printTask` lambda (lines 23–52). Each **`jthread`** passes its **`stop_token`** to the lambda. Lines 36–48 in the lambda loop continuously

- printing the task name, thread ID and sleep time,
- sleeping, then
- printing the task name and saying that the lambda is performing work.

The loop executes until the corresponding `jthread`'s internal `stop_source` receives a call to its `request_stop` member function. In this program, that call occurs in the `jthread`'s destructor, when the `jthreads` go out of scope at the end of `main`.

Optional `stop_callback`

Lines 31–34 also demonstrate that **you can register an optional function to call when a stop is requested**. The `std::stop_callback`[68] constructor receives as arguments a `stop_token` and a function with no parameters—in this case, a lambda. The constructor registers the function with the given `stop_token`. When the `stop_token` is notified that a stop was requested, it calls the function registered by the `stop_callback` on the thread that requested the stop (the `main` thread in this example). Any number of `stop_callbacks` can be created for a given `stop_token`. **The order in which their functions execute is not specified**. Our lambda simply displays a `string` to show that the callback was indeed called, but this could be used to perform cleanup operations before a thread terminates.[69]

68. "`std::stop_callback`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/stop_callback.
69. Williams, "Concurrency in C++20 and Beyond."

Analyzing the Output

Throughout the sample output, `Task 1` and `Task 2` sleep and work. Once `main` ends, the `jthreads` executing these tasks go out of scope, and their destructors call each `jthread`'s `request_stop` member function to notify the tasks that they should terminate. In our output after `"MAIN (id: 16352) ENDS"`, you can see when each `jthread` received its `request_stop` call—our `stop_callback` displays the task name followed by a message that includes the ID of the thread that told the task to stop (the `main` thread in this example).[70] In each case, the corresponding task then finishes its work and terminates.

70. If the stop is requested before the `stop_callback` is constructed, then the thread ID displayed will that of the thread constructing the `stop_callback` (per https://en.cppreference.com/w/cpp/thread/stop_callback).

17.10 Launching Tasks with `std::async`

Figure 17.12 demonstrates `std::async`—**a higher-level way to launch a task in a separate thread**. In this example, we'll implement a potentially long-running task—determining whether a large integer is prime and, if not, determining its prime factors.

Security, Encryption and Prime Factorization for Enormous Integers

Sec Security is a crucial application. Prime factorization[71,72,73] is a vital aspect of the **RSA Public-Key Cryptography algorithm**, which is commonly used to secure data transmitted over the Internet.[74,75,76] Industrial-strength RSA works with enormous prime numbers consisting of hundreds of digits. The sheer amount of time required to factor the product of those primes—even for the most powerful supercomputers in use today—is a key reason why RSA is so secure. Public-key cryptography also is used to secure the blockchain technology behind cryptocurrencies, like Bitcoin.[77] If you're interested in learning how RSA works, see the online RSA Public-Key Cryptography appendix in the Resources section of the book's webpage:

71. “Prime Factorization,” July 2, 2020. Accessed February 7, 2022. https://www.cuemath.com/numbers/prime-factorization/.
72. Striver, “Prime Factors of a Big Number,” May 28, 2021. Accessed February 7, 2022. https://www.geeksforgeeks.org/prime-factors-big-number/.
73. Ehud Shalit, “Prime Numbers—Why Are They So Exciting?” September 7, 2018. Accessed February 7, 2022. https://kids.frontiersin.org/articles/10.3389/frym.2018.00040.
74. “RSA (Cryptosystem).” Accessed February 7, 2022. https://en.wikipedia.org/wiki/RSA_(cryptosystem).
75. “RSA Algorithm.” Accessed February 7, 2022. https://simple.wikipedia.org/wiki/RSA_algorithm.
76. K. Moriarty, B. Kaliski, J. Jonsson and A. Rusch, “PKCS #1: RSA Cryptography Specifications Version 2.2,” November 2016. Accessed February 7, 2022. https://tools.ietf.org/html/rfc8017.
77. Sarah Rothrie, “How Blockchain Cryptography Is Fighting the Rise of Quantum Machines,” December 28, 2018. Accessed February 7, 2022. https://coincentral.com/blockchain-cryptography-quantum-machines/.

Overview of This Example

This example’s task to execute is defined by function `getFactors` (lines 27–68), which determines whether a number is prime[78] and, if not, determines its prime factors. Each thread in this program executes until `getFactors` returns its result—a `std::tuple` containing the task name, the number `getFactors` processed, a `bool` indicating whether the number is prime and a `vector` of the number’s factors. To ensure that our tasks run for at least a few seconds each, we used two 19-digit numbers, including a prime value from the University of Tennessee Martin’s Prime Pages website.[79] Its prime-number research includes lists of prime values by their numbers of digits.

78. “Primality Test.” Accessed February 7, 2022. https://en.wikipedia.org/wiki/Primality_test.
79. “Index: Numbers.” Accessed February 7, 2022. https://primes.utm.edu/curios/index.php.

Figure 17.12 consists of the following functions:

- Function `id` (lines 13–17) converts a **unique** `std::thread::id` to a `std::string`.
- Function `getFactors` (lines 27–68) attempts to find an integer’s factors.
- Function `proveFactors` (lines 71–90) confirms that a given set of prime factors, when multiplied together, reproduces a corresponding non-prime value.
- Function `displayResults` (lines 93–117) displays a task’s results.
- Function `main` (lines 119–133) launches the `getFactors` tasks, waits for them to complete, then displays their results.

We’ve split this program into pieces for discussion purposes. After the program, we show sample outputs. In Fig. 17.12, lines 3–10 `import` the headers used in this program.

```
1  // Fig. 17.12: async.cpp
2  // Prime-factorization tasks performed in separate threads
3  #include <cmath>
4  #include <fmt/format.h>
5  #include <future> // std::async
6  #include <iostream>
7  #include <sstream>
8  #include <string>
```

```
 9   #include <tuple>
10   #include <vector>
11
12   // get current thread's ID as a string
13   std::string id() {
14      std::ostringstream out;
15      out << std::this_thread::get_id();
16      return out.str();
17   }
```

Fig. 17.12 Prime-factorization tasks launched with `std::async`.

Type Aliases

Lines 20 and 24 define type aliases we use to simplify type declarations:

- `Factors` (line 20) is an alias for a `vector` of `pair`s, each containing a prime factor and how many times an integer was divisible by that factor. For example, 8 has three factors of 2.
- `FactorResults` (line 24) is the `getFactors` function's return type. This alias represents a `tuple` that contains a task's name (`std::string`), the number for which we'll determine the prime factors (`long long`), whether the number is prime (`bool`) and the prime factors (`Factors`).

```
19   // type alias for vector of factor/count pairs
20   using Factors = std::vector<std::pair<long long, int>>;
21
22   // type alias for a tuple containing a task name,
23   // a number, whether the number is prime and its factors
24   using FactorResults = std::tuple<std::string, long long, bool, Factors>;
25
```

`getFactors` Function to Determine Prime Factorization of an Integer

This program launches separate threads that each call function `getFactors` (lines 27–68) to find an integer's factors or determine that it's prime. The function receives a task name to display in outputs and the number to factor and returns a `FactorResults` object.

```
26   // performs prime factorization
27   FactorResults getFactors(std::string name, long long number) {
28      std::cout << fmt::format(
29         "{}: Thread {} executing getFactors for {}\n", name, id(), number);
30
31      long long originalNumber{number}; // copy to place in FactorResults
32      Factors factors; // vector of factor/count pairs
33
34      // lambda that divides number by a factor and stores factor/count
35      auto factorCount{
36         [&](int factor) {
37            int count{0}; // how many times number is divisible by factor
38
39            // count how many times number is divisible by factor
40            while (number % factor == 0) {
41               ++count;
```

```
42                number /= factor;
43             }
44
45             // store pair containing the factor and its count
46             if (count > 0) {
47                factors.push_back({factor, count});
48             }
49          }
50       };
51
52       factorCount(2); // count how many times number is divisible by 2
53
54       // number is now odd; store each factor and its count
55       for (int i{3}; i <= std::sqrt(number); i += 2) {
56          factorCount(i); // count how many times number is divisible by i
57       }
58
59       // add last prime factor
60       if (number > 2) {
61          factors.push_back({number, 1});
62       }
63
64       bool isPrime{factors.size() == 1 && get<int>(factors[0]) == 1};
65
66       // initialize the FactorResults object returned by getFactors
67       return {name, originalNumber, isPrime, factors};
68    }
69
```

The function operates as follows:

- Lines 28–29 display the executing task's `name` and the ID of the thread in which `getFactors` is executing.
- If `number` has prime factors, this algorithm will modify `number`, so line 31 copies `number` for inclusion in the `FactorResults`.
- Line 32 defines the `Factors` object—a `vector` of factor/count `pair`s.
- Lines 35–50 define the lambda `factorCount`, which counts how many times `number` (which is captured by reference) is divisible by the lambda's `factor` argument. While `number` is divisible by `factor` (line 40), we increment the count and divide `number` by `factor`. Then, if the `count` is greater than 0, line 47 adds a new `factor`/`count` pair to the `factors` object.
- Line 52 calls `factorCount` for the factor `2`. Then, lines 55–57 repeatedly call it for the odd values `3` and above while `i` is less than or equal to `number`'s square root.
- Lines 60–62 check whether `number` is still greater than `2`. If so, this is the last factor to add to `factors`.
- If `number` is prime, `factors` will contain only the number itself and its factor count will be `1`. Line 64 checks this and sets `bool` variable `isPrime` accordingly.
- Finally, line 67 initializes the `FactorResults tuple` that `getFactors` returns, using the task `name`, `originalNumber`, `isPrime` and `factors`.

`proveFactors` Function to Confirm Prime Factorization

Lines 71–90 define `proveFactors`, which confirms that a non-prime integer value can be reproduced by calculating the product of its prime factors:

- Line 72 initializes `proof` to `1`.

- For each prime `factor`/`count pair` in `factors` (line 76), lines 77–79 iterate `count` times, multiplying `proof` by that `factor`.
- Lines 83–89 check if `number` matches `proof` and display an appropriate message.

```
70   // multiply the factors and confirm they reproduce number
71   void proveFactors(long long number, const Factors& factors) {
72      long long proof{1};
73
74      // for each factor/count pair, unpack it then multiply proof
75      // by factor the number of times specified by count
76      for (const auto& [factor, count] : factors) {
77         for (int i{0}; i < count; ++i) {
78            proof *= factor;
79         }
80      }
81
82      // confirm proof and number are equal
83      if (proof == number) {
84         std::cout << fmt::format(
85            "\nProduct of factors matches original value ({})\n", proof);
86      }
87      else {
88         std::cout << fmt::format("\n{} != {}\n", proof, number);
89      }
90   }
91
```

`displayResults` Function to Display Prime Factorization

Lines 93–117 define `displayResults`, which `main` calls to display each `FactorResults tuple` received from this program's tasks:

- Line 96 unpacks the `tuple` into variables `name` (`std::string`), `number` (`long long`), `isPrime` (`bool`) and `factors` (`Factors`). Each variable's type is inferred from the tuple's type (line 24).
- Line 98 displays the task `name`.
- If the number is prime (line 101), line 102 displays a message. Otherwise, lines 105–110 display the non-prime number's prime factors and their counts.
- Finally, if the `number` is not prime, line 115 calls `proveFactors` to check whether the prime `factors`, when multiplied, reproduce the `number`.

```
92   // show a task's FactorResults
93   void displayResults(const FactorResults& results) {
94      // unpack results into name (std::string), number (long long),
95      // isPrime (bool) and factors (Factors)
96      const auto& [name, number, isPrime, factors] {results};
97
98      std::cout << fmt::format("\n{} results:\n", name);
99
100      // display whether value is prime
101      if (isPrime) {
102         std::cout << fmt::format("{} is prime\n", number);
103      }
```

```
104      else { // display prime factors
105         std::cout << fmt::format("{}'s prime factors:\n\n", number);
106         std::cout << fmt::format("{:<12}{:<8}\n", "Factor", "Count");
107
108         for (const auto& [factor, count] : factors) {
109            std::cout << fmt::format("{:<12}{:<8}\n", factor, count);
110         }
111      }
112
113      // if not prime, prove that factors produce the original number
114      if (!isPrime) {
115         proveFactors(number, factors);
116      }
117   }
118
```

Creating and Executing a Task: Function template `std::async`

The `<future>` header (line 5) contains features that enable you to **execute asynchronous tasks** and **receive the results of those tasks** when they finish executing.

```
119   int main() {
120      std::cout << "MAIN LAUNCHING TASKS\n";
121      auto future1{std::async(std::launch::async,
122         getFactors, "Task 1", 1016669006116682993)}; // not prime
123      auto future2{std::async(std::launch::async,
124         getFactors, "Task 2", 1000000000000000003)};// prime
125
126      std::cout << "\nWAITING FOR TASK RESULTS\n";
127
128      // wait for results from each task, then display the results
129      displayResults(future1.get());
130      displayResults(future2.get());
131
132      std::cout << "\nMAIN ENDS\n";
133   }
```

```
MAIN LAUNCHING TASKS

WAITING FOR TASK RESULTS
Task 1: Thread 5032 executing getFactors for 1016669006116682993
Task 2: Thread 2952 executing getFactors for 1000000000000000003

Task 1 results:
1016669006116682993's prime factors:

Factor      Count
1000000007  1
1016668999  1

Product of factors matches original value (1016669006116682993)

Task 2 results:
1000000000000000003 is prime
```

```
MAIN ENDS
```

11 Lines 121–122 and 123–124 use the **`std::async` function template**[80] to create two threads that execute `getFactors` asynchronously. Function `std::async` has two versions. The one used here takes three arguments:

80. "`std::async`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/async.

- The **launch policy** (from the `std::launch` `enum`) is `std::launch::async`, `std::launch::deferred` or both separated by a bitwise OR (`|`) operator. The value `std::launch::async` indicates that the function specified in the second argument should execute in a separate thread, whereas `std::launch::deferred` indicates that it should execute in the same thread. Combining these values lets the system choose whether to execute asynchronously or synchronously.
- The task to execute is specified by a **function pointer**, **function object** or **lambda**.
- Any remaining arguments are passed by `std::async` to the task function. We passed a string name for the task and a `long long` value specifying the number for which to calculate the prime factorization.

The other `std::async` version does not receive a launch policy argument—it chooses the launch policy for you.

If `std::async` receives the launch policy `std::launch::async` but cannot create the thread, it throws a `std::system_error` exception. If `std::async` creates the thread successfully, the task function begins executing in the new thread.

`std::future`, `std::promise` and Inter-Thread Communication

11 CG Function `std::async` returns an object of class template `std::future` (C++11), which enables **inter-thread communication** between the thread that calls `async` and the task `async` executes. The C++ Core Guidelines recommend using a `future` to return a result from an asynchronous task.[81]

81. C++ Core Guidelines, "CP.60: Use a `future` to Return a Value from a Concurrent Task." Accessed February 7, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rconc-future.

"Under the hood," `async` uses a `std::promise` object from which it obtains the returned `future`. When the task completes, `async` stores the task's result in the `promise`. `async`'s caller uses the `future` to access the result in the `promise`—in our case, a `Factor-Results` object. You do not need to work with the `promise` directly.

To ensure that the program does not terminate until the tasks complete and to receive the results from each task, lines 129–130 call each `future`'s `get` member function. If the task is still running, `get` blocks the calling thread, which **waits** until the task completes; otherwise, `get` immediately returns the task's result. Function `get`'s return type is whatever the task function returns. Once the results are available, lines 129–130 pass them to function `displayResults`. The program's output shows unique thread IDs, confirming that **both `getFactors` tasks executed in separate threads**.

Err If `async`'s task throws an exception, the `future` contains the exception rather than the task's result, and **calling `get` rethrows the exception**. If multiple threads need access to an asynchronous task's result, you can use `std::shared_future` objects.[82]

82. "`std::shared_future`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/shared_future.

`std::packaged_task` vs. `std::async`

Another way to launch asynchronous tasks is `std::packaged_task`.[83] The key differences between `async` and `packaged_task` are as follows:

83. "`std::packaged_task`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/packaged_task.

- You must call its `get_future` member function to get the associated `future` object for obtaining the task's results at a later time.
- A `packaged_task` executes on the thread that calls the task's `operator()` function.
- To execute a `packaged_task` in a separate thread, you create the thread and initialize it with `std::move(`*yourPackagedTaskObject*`)`. When the thread completes, you can call `get` on the task's `future` object to obtain the result.

17.11 Thread-Safe, One-Time Initialization

11 Sometimes a variable should be initialized exactly once, even when concurrent threads try to execute the variable's initialization statement. Before C++11, a technique called double-checked locking was commonly used, but it was not guaranteed to be thread-safe across all compilers and platforms.[84] Here, we discuss two mechanisms C++11 provides to perform **one-time, thread-safe initialization of a variable**.

84. Scott Meyers and Andrei Alexandrescu, "C++ and the Perils of Double-Checked Locking," September 2004. Accessed February 7, 2022. https://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf.

`static` Local Variables

11 When a function defines a `static` local variable and concurrent threads execute that function, the C++ standard specifies that the variable's initialization "is performed the first time control passes through its declaration."[85, 86] So, the first thread that executes the `static` local variable's declaration performs the initialization. All other threads attempting to execute the `static` variable's declaration are **blocked** until the first thread completes the variable's initialization. Then the other threads calling the function skip the declaration and use the already initialized `static` local variable.

85. C++ Standard, "8.8 Declaration Statement." Accessed February 7, 2022. https://timsongcpp.github.io/cppwp/n4861/stmt.dcl#4.
86. "Storage Class Specifiers—`static` Local Varables." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/language/storage_duration#Static_local_variables.

11 `once_flag` and `call_once`

11 C++11's `std::once_flag`[87] and `std::call_once`[88] (from the `<mutex>` header) are used together to ensure that concurrent threads execute a function, lambda or function object exactly once. First, declare a `once_flag` object:

87. "`std::once_flag`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/once_flag.
88. "`std::call_once`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/call_once.

```
std::once_flag myFlag;
```

Then, use `call_once` to call your function:

```
std::call_once(myFlag, myFunction, arguments);
```

The first thread that executes the preceding statement will call *myFunction*, passing the specified *arguments* (if any). When a subsequent thread executes this statement, `call_once` returns immediately and does not call *myFunction*. For a common `call_once` use-case, see Arthur O'Dwyer's "Back to Basics: Concurrency" video.[89]

89. Arthur O'Dwyer, "Back to Basics: Concurrency," October 6, 2020. Accessed February 7, 2022. https://www.youtube.com/watch?v=F6Ipn7gCOsY.

17.12 A Brief Introduction to Atomics

11 The examples in Sections 17.6–17.7 ensured exclusive access to shared mutable data using the synchronization primitives `std::mutex`, **locks** and `std::condition_variable`. C++11 also introduced **`atomic` types**[90] (from the **`<atomic>` header**[91]), which provide operations that cannot be divided into smaller steps, enabling threads to conveniently **share mutable data without explicit synchronization and locking**.

90. Hans-J. Boehm and Lawrence Crowl, "C++ Atomic Types and Operations," October 3, 2007. Accessed February 7, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2427.html.
91. "Atomic Operations Library." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/atomic.

20 In a sense, atomics are higher level than **`mutex`es**, **locks** and **`condition_variable`s**. However, atomic types and their operations are primarily considered to be **low-level features** intended to help library developers implement **higher-level concurrency capabilities**. For example, the Visual C++, GNU C++ and Clang C++ standard libraries each use atomics "under the hood" to implement **C++20's `std::latch`**, **`std::barrier`** and **`std::semaphore`** **thread-coordination primitives** (Sections 17.13–17.14).[92,93,94]

92. "Microsoft's C++ Standard Library." Accessed February 7, 2022. https://github.com/microsoft/STL.
93. "libstdc++." Accessed February 7, 2022. https://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/files.html.
94. "libc++ Documentation." Accessed February 7, 2022. https://github.com/llvm/llvm-project/tree/main/libcxx/.

We present simple examples of atomics. Most developers will prefer higher-level primitives, which help them quickly build correct, robust, efficient concurrent applications. For an extensive discussion of atomics' performance vs. that of related technologies, see the blog post, "A Concurrency Cost Hierarchy."[95]

95. Travis Downs, "A Concurrency Cost Hierarchy," July 6, 2020. Accessed February 7, 2022. https://travisdowns.github.io/blog/2020/07/06/concurrency-costs.html.

Incrementing an `int`, a `std::atomic<int>` and a `std::atomic_ref<int>` from Two Concurrent Threads

Figure 17.13 presents a simple demonstration of two concurrent threads incrementing

- an unprotected `int`,
- an object of type `std::atomic<int>` and
- an object of the **class template** `std::atomic_ref<int>`.

Class template `std::atomic_ref` was introduced in C++20 to enable **atomic operations on a referenced variable** (an `int` in this program). First, we'll show `std::atomic<int>` and `std::atomic_ref<int>` in action, then we'll say a bit more about atomics.

```
// Fig. 17.13: atomic.cpp
// Incrementing integers from concurrent threads
// with and without atomics.
#include <atomic>
#include <fmt/format.h>
#include <iostream>
#include <thread>

int main() {
   int count1{0};
   std::atomic<int> atomicCount{0};
   int count2{0};
   std::atomic_ref<int> atomicRefCount{count2};

   {
      std::cout << "Two concurrent threads incrementing int count1, "
         << "atomicCount and atomicRefCount\n\n";

      // lambda to increment counters
      auto incrementer{
         [&]() {
            for (int i{0}; i < 1000; ++i) {
               ++count1; // no synchronization
               ++atomicCount; // ++ is an atomic operation
               ++atomicRefCount; // ++ is an atomic operation
               std::this_thread::yield(); // force thread to give up CPU
            }
         }
      };

      std::jthread t1{incrementer};
      std::jthread t2{incrementer};
   }

   std::cout << fmt::format("Final count1: {}\n", count1);
   std::cout << fmt::format("Final atomicCount: {}\n", atomicCount);
   std::cout << fmt::format("Final count2: {}\n", count2);
}
```

```
Two concurrent threads incrementing int count1, atomicCount and
atomicRefCount

Final count1: 2000
Final atomicCount: 2000
Final count2: 2000
Two concurrent threads incrementing int count1, atomicCount and
atomicRefCount

Final count1: 1554
Final atomicCount: 2000
```

```
Final count2: 2000
```

Fig. 17.13 Incrementing integers from concurrent threads with and without atomics.

Defining the Counters

Lines 10–13 define

- `int` variable `count1`,
- `std::atomic<int>` variable `atomicCount`,
- `int` variable `count2` and
- `std::atomic_ref<int>` variable `atomicRefCount`.

The first three variables are initialized to `0`, and `atomicRefCount` is initialized with a reference to the variable `count2`. We'll use `atomicRefCount` to show that **you can indirectly manipulate `count2` atomically through a reference**.

Incrementing the Counters

SE Lines 15–33 use two **unsynchronized concurrent threads** to increment the counters 1,000 times per thread. If every increment works correctly, each counter's total should be 2,000 when these threads complete. Lines 20–29 define the lambda `incrementer`, which uses the operator `++` to increment each of `count1` (line 23), `atomicCount` (line 24) and `atomicRef-Count` (line 25). **Operator `++` is one of a limited number of operations you can perform on atomic objects. When one thread is executing `++` on an atomic object, the increment is guaranteed to complete before another thread can modify or view that atomic object's value.** The same is not guaranteed for a non-atomic `int` variable. For a complete list of atomic operations, see

```
https://en.cppreference.com/w/cpp/atomic/atomic
```

At today's processor speeds, a thread can perform loop iterations quickly, so multiple threads incrementing an `int` **without synchronization** might appear to work correctly, as shown in this program's first output. For this reason, line 26 uses the `std::this_thread` namespace's `yield` function to **force the currently executing thread to relinquish the processor after each loop iteration**. Like sleeping, yielding helps us emphasize that **you cannot predict the relative speeds of asynchronous concurrent threads**.

Lines 31–32 create `std::jthreads` that each execute `incrementer`. Then, the block terminates, and the `jthreads` go out of scope, automatically joining the threads. Finally, lines 35–37 display the values of `count1`, `atomicCount` and `count2`. The variable `count2` was incremented indirectly via `atomicRefCount`.

Analyzing the Sample Outputs

Err Lines 20–29 should increment the `count1`, `atomicCount` and `count2` (indirectly through `atomicRefCount`) 2,000 times. However, the sample outputs show that incrementing the `int` `count` from concurrent threads **without synchronization** can produce different totals each time, including the seemingly correct total in the first output. Incrementing `atomic-Count` and incrementing `count2` indirectly through `atomicRefCount` from concurrent threads **without synchronization** is guaranteed to produce the proper total of 2,000 for each.

More About Atomic Types[96, 97]

96. C++ Standard, “Atomic Operations Library.” Accessed February 7, 2022. https://timsongcpp.github.io/cppwp/n4861/atomics.
97. “`std::atomic`.” Accessed February 7, 2022. https://en.cppreference.com/w/cpp/atomic/atomic.

There are predefined **`std::atomic`** class template specializations and corresponding type aliases for type `bool` and every integral type. You also can specialize **`std::atomic`** for

- pointer types,
- floating-point types and
- **trivially copyable types**[98]—that is, the compiler provides the copy and move constructors, copy and move assignment operators, and destructor, and there are no `virtual` functions.

98. C++ Standard, “Properties of Classes.” Accessed February 7, 2022. https://timsongcpp.github.io/cppwp/n4861/class.prop.

20 The specializations for integral, pointer and, as of C++20, floating-point types each support adding a value to or subtracting a value from an atomic data item. The integral specializations also provide atomic bitwise **`&=`** (**and**), **`|=`** (**or**) and **`^=`** (**xor**) operations.

C++20 Atomic Smart Pointers and Atomic Pointers[99, 100]

99. C++ Standard, “Partial Specializations for Smart Pointers.” Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/util.smartptr.atomic.
100. “`std::atomic`.” Accessed February 7, 2022. https://en.cppreference.com/w/cpp/atomic/atomic.

20 C++20 adds atomic smart pointer specializations for **`std::atomic<shared_ptr<T>>`** and **`std::atomic<weak_ptr<T>>`**, enabling atomic pointer manipulations of these smart pointer types.

For all **atomic pointers**, only the pointer manipulations are atomic—for example,

- aiming the pointer at a different object,
- incrementing the pointer by an integer or
- decrementing the pointer by an integer.

Atomic pointers can be used to implement **thread-safe, linked data structures**.

20 C++20 `std::atomic_ref` Class Template

Err ⊗ Like **`std::atomic`**, the **`std::atomic_ref`** class template can be specialized for any primitive type, pointer type or trivially copyable type. However, a **`std::atomic_ref`** object is **initialized with a reference** rather than a value. You use the **`atomic_ref`** to perform atomic operations on the referenced object. Multiple **`atomic_ref`**s can refer to the same object. The referenced object must outlive the **`std::atomic_ref`**; otherwise, a dangling reference or pointer will result.[101, 102]

101. C++ Standard, “31.7 Class Template `atomic_ref`.” Accessed February 7, 2022. https://timsongcpp.github.io/cppwp/n4861/atomics.ref.generic.
102. “`std::atomic_ref`.” Accessed February 7, 2022. https://en.cppreference.com/w/cpp/atomic/atomic_ref.

20 17.13 Coordinating Threads with C++20 Latches and Barriers

20 So far, we’ve coordinated threads using **`std::mutex`**, **`std::condition_variable`** and locks. C++20 provides higher-level thread-coordination types **`std::latch`** and **`std::barrier`**,[103] which do not require explicit use of mutexes, condition variables and locks.[104]

103. C++ Standard, “Thread Support Library—Coordination Types,” Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/thread.coord.
104. Bryce Adelstein Lelbach, “The C++20 Synchronization Library,” October 24, 2019. November 30, 2021. https://www.youtube.com/watch?v=Zcqwb3CWqs4.

20 17.13.1 C++20 `std::latch`

A **`std::latch`** (from the **`<latch>` header**) is a **single-use gateway** in your code that remains closed until a specified number of threads reach the latch. At that point, the gateway remains open permanently.[105] The gateway serves as **a one-time synchronization point**, allowing threads to wait until a specified number of threads reach that point. **Latches are simpler to use than mutexes and condition variables for coordinating threads**.
105. C++ Standard, “Thread Support Library—Coordination Types—Latches.” Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/thread.latch.

Consider a parallel sorting algorithm that

- launches several worker threads to sort portions of a large `array`,
- waits for the worker threads to complete, then
- merges the sorted sub-`array`s into the final sorted `array`.

Assume the algorithm uses two worker threads, each sorting half the `array`. The algorithm can use a **`std::latch`** object to wait until the workers are done:

- First, the algorithm creates a **`std::latch`** with a **non-zero count**. In this case, it needs two worker threads to complete, so we initialize the **`latch`** to `2`.
- Each worker has a reference to the same **`latch`**.
- After launching the workers, the algorithm **waits on that `latch`**, which **blocks the algorithm from continuing**.
- When a thread reaches the **`latch`**, the thread reduces the **`latch`**'s count by `1` (known as **signaling the latch**) to indicate that it finished sorting a sub-`array`.
- When the **`latch`**'s count becomes `0`, the **`latch`** **permanently opens**, **unblocking** the algorithm's thread so it can merge the results of its worker threads.

Attempting to Wait When the Gateway Is Permanently Open

Any number of threads can attempt to wait on a given **`latch`**. However, once the **`latch`** is open, other threads that attempt to wait on it simply pass through the gateway and continue executing.

Demonstrating Latches

Figure 17.14 uses a lambda named `task` (lines 22–34) to perform work that must complete before `main` is allowed to continue executing. As we've done throughout this chapter, we simulate the work by sleeping. The program operates as follows:

- Line 19 creates the **`std::latch`** `mainLatch` with the count `3`. Any thread(s) waiting on `mainLatch` cannot continue executing until three threads reach `mainLatch`.
- Lines 40–46 launch three **`jthread`s**—each executes the `task` lambda.
- After `main` launches the **`jthread`s**, line 49 calls the `mainLatch`'s **`wait` member function**. If `mainLatch`'s count is greater than `0`, `main` waits at line 49 until the latch's count reaches `0`.
- When each **`jthread`**'s `task` call reaches line 32, it calls `mainLatch`'s **`count_down` member function** to **signal the latch**, decrementing its count. Though the `task` terminates in this program, it could continue working. The key is that the work `main` is waiting for should be completed by each

thread before it **signals the latch**. In the outputs, note that the threads signal the latch in different orders.

- When `mainLatch`'s count reaches `0`, any thread(s) waiting on `mainLatch` unblock and continue executing. In this program, `main` continues executing at line 51.
- To show that a latch is a **single-use gateway**, line 53 calls `mainLatch`'s **`wait`** function again. Since `mainLatch` is already open, `main` simply continues executing at line 54, displays another message then terminates.

```
 1  // Fig. 17.14: LatchDemo.cpp
 2  // Coordinate threads with a std::latch object.
 3  #include <chrono>
 4  #include <fmt/format.h>
 5  #include <iostream>
 6  #include <latch>
 7  #include <random>
 8  #include <string_view>
 9  #include <thread>
10  #include <vector>
11
12  int main() {
13     // set up random-number generation
14     std::random_device rd;
15     std::default_random_engine engine{rd()};
16     std::uniform_int_distribution ints{2000, 3000};
17
18     // latch that 3 threads must signal before the main thread continues
19     std::latch mainLatch{3};
20
21     // lambda representing the task to execute
22     auto task{
23        [&](std::string_view name, std::chrono::milliseconds workTime) {
24           std::cout << fmt::format("Proceeding with {} work for {} ms.\n",
25              name, workTime.count());
26
27           // simulate work by sleeping
28           std::this_thread::sleep_for(workTime);
29
30           // show that task arrived at mainLatch
31           std::cout << fmt::format("{} done; signals mainLatch.\n", name);
32           mainLatch.count_down();
33        }
34     };
35
36     std::vector<std::jthread> threads; // stores the threads
37     std::cout << "Main starting three jthreads.\n";
38
39     // start three jthreads
40     for (int i{1}; i < 4; ++i) {
41        // create jthread that calls task lambda,
42        // passing a task name and work time
43        threads.push_back(std::jthread{task,
44           fmt::format("Task {}", i),
```

```
            std::chrono::milliseconds{ints(engine)}});
      }

      std::cout << "\nMain waiting for jthreads to reach the latch.\n\n";
      mainLatch.wait();

      std::cout << "\nAll jthreads reached the latch. Main working.\n";
      std::cout << "Showing that mainLatch is permanently open.\n";
      mainLatch.wait(); // latch is already open
      std::cout << "mainLatch is already open. Main continues.\n";
   }
```

```
Main starting three jthreads.

Main waiting for jthreads to reach the latch.

Proceeding with Task 3 work for 2648 ms.
Proceeding with Task 2 work for 2705 ms.
Proceeding with Task 1 work for 2024 ms.
Task 1 done; signals mainLatch.
Task 3 done; signals mainLatch.
Task 2 done; signals mainLatch.

All jthreads reached the latch. Main working.
Showing that mainLatch is permanently open.
mainLatch is already open. Main continues.
Main starting three jthreads.

Main waiting for jthreads to reach the latch.

Proceeding with Task 2 work for 2571 ms.
Proceeding with Task 1 work for 2462 ms.
Proceeding with Task 3 work for 2248 ms.
Task 3 done; signals mainLatch.
Task 1 done; signals mainLatch.
Task 2 done; signals mainLatch.

All jthreads reached the latch. Main working.
Showing that mainLatch is permanently open.
mainLatch is already open. Main continues.
```

Fig. 17.14 Coordinating threads with a `std::latch` object.

17.13.2 C++20 `std::barrier`

20 Consider a simulation of the painting step in an automated automobile assembly line. Often several computer-controlled robots work together to perform a given step. Let's assume that cars moving along the assembly line and two robots' operations are all individually controlled by threads. Once the work on one car finishes, we want to reset everything, advance the assembly line and perform the work again for the next car. The preceding scenario is ideal for a **`std::barrier`**[106] (from the **`<barrier>` header**), which is like a **reusable `latch`**. Typically, a **`barrier`** is used for repetitive tasks in a loop:

106. C++ Standard, "Thread Support Library—Coordination Types—Barriers." Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/thread.barrier.

- Each thread works then reaches a **`barrier`** and waits for it to open.

- When the specified number of threads reaches the `barrier`, an optional completion function executes.
- The `barrier` resets its count, which unblocks the threads so they may continue executing and repeat this process.

Using a `barrier` (Fig. 17.15), let's simulate an assembly line's painting step for multiple cars. We'll also use `stop_source` and `stop_token` manually to coordinate thread cancellation when the assembly line shuts down. Again, we'll use sleeping to simulate work. Note in the output that the robots sometimes finish their "work" on each car in a different order.

```
1  // Fig. 17.15: BarrierDemo.cpp
2  // Coordinating threads with a std::barrier object.
3  #include <barrier>
4  #include <chrono>
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <random>
8  #include <string_view>
9  #include <thread>
10
11 int main() {
12    // simulate moving car into painting position
13    auto moveCarIntoPosition{
14       []() {
15          std::cout << "Moving next car into painting position.\n";
16          std::this_thread::sleep_for(std::chrono::seconds(1));
17          std::cout << "Car ready for painting.\n\n";
18       }
19    };
20
21    int carsToPaint{3};
22
23    // stop_source used to notify robots assembly line is shutting down
24    std::stop_source assemblyLineStopSource;
25
26    // stop_token used by paintingRobotTask to determine when to shut down
27    std::stop_token stopToken{assemblyLineStopSource.get_token()};
28
29    // assembly line waits for two painting robots to reach this barrier
30    std::barrier paintingDone{2,
31       [&]() noexcept { // lambda called when robots finish
32          static int count{0}; // # of cars that have been painted
33          std::cout << "Painting robots completed tasks\n\n";
34
35          // check whether it's time to shut down the assembly line
36          if (++count == carsToPaint) {
37             std::cout << "Shutting down assembly line\n\n";
38             assemblyLineStopSource.request_stop();
39          }
40          else {
41             moveCarIntoPosition();
42          }
43       }
```

```
44      };
45
46      // lambda that simulates painting work
47      auto paintingRobotTask{
48         [&](std::string_view name) {
49            // set up random-number generation
50            std::random_device rd;
51            std::default_random_engine engine{rd()};
52            std::uniform_int_distribution ints{2500, 5000};
53
54            // check whether the assembly line is shutting down
55            // and, if not, do the painting work
56            while (!stopToken.stop_requested()) {
57               auto workTime{std::chrono::milliseconds{ints(engine)}};
58
59               std::cout << fmt::format("{} painting for {} ms\n",
60                  name, workTime.count());
61               std::this_thread::sleep_for(workTime);// simulate work

63               // show that task woke up and arrived at continuationBarrier
64               std::cout << fmt::format(
65                  "{} done painting. Waiting for next car.\n", name);
66
67               // decrement paintingDone barrier's counter and
68               // wait for other painting robots to arrive here
69               paintingDone.arrive_and_wait();
70            }
71
72            std::cout << fmt::format("{} shut down.\n", name);
73         }
74      };
75
76      moveCarIntoPosition(); // move the first car into position
77
78      // start up two painting robots
79      std::cout << "Starting robots.\n\n";
80      std::jthread leftSideRobot{paintingRobotTask, "Left side robot"};
81      std::jthread rightSideRobot{paintingRobotTask, "Right side robot"};
82   }
```

```
Moving next car into painting position.
Car ready for painting.

Starting robots.

Left side robot painting for 4564 ms
Right side robot painting for 2758 ms
Right side robot done painting. Waiting for next car.
Left side robot done painting. Waiting for next car.
Painting robots completed tasks

Moving next car into painting position.
Car ready for painting.

Left side robot painting for 4114 ms
Right side robot painting for 2860 ms
```

```
Right side robot done painting. Waiting for next car.
Left side robot done painting. Waiting for next car.
Painting robots completed tasks

Moving next car into painting position.
Car ready for painting.

Right side robot painting for 4730 ms
Left side robot painting for 3794 ms
Left side robot done painting. Waiting for next car.
Right side robot done painting. Waiting for next car.
Painting robots completed tasks

Shutting down assembly line

Right side robot shut down.
Left side robot shut down.
```

Fig. 17.15 Coordinating threads with `std::barrier` objects.

The `main` thread represents the assembly line. The program operates as follows:

- Lines 13–19 define a lambda to simulate moving the next car into the assembly line's painting station.
- Line 21 defines `carsToPaint`—the number of cars to process in the simulation.
- We use **cooperative cancellation** (Section 17.9) to terminate the robot threads when the assembly line shuts down. `jthread`'s destructor calls its `stop_source`'s `request_stop` member function before joining the thread. The `jthread`'s task can then check its `stop_token` parameter to determine whether to terminate. In this simulation, however, we do not want the robot threads to terminate when the `jthread`s go out of scope at the end of `main`, so we handle the cancellation manually. The `stop_source` (line 24) coordinates cancellation with the robot threads. When three cars have been processed, we'll call `request_stop` on this `stop_source` to notify the robot threads that the assembly line is shutting down.
- Line 27 gets the `stop_source`'s `stop_token`. Before painting, each robot thread will call this `stop_token`'s `stop_requested` function to check whether the assembly line is shutting down.
- Lines 30–44 define the `paintingDone` `barrier`, initializing it with a count of `2` (for the two painting robots) and a **completion function** (lines 31–43) that's called when the `barrier`'s internal count reaches `0`. The `static` local variable `count` tracks the number of cars processed so far. When the count equals `carsToPaint`, line 38 calls `request_stop` on the `stop_source` to indicate the assembly line is shutting down. Otherwise, line 41 simulates moving the next car into position.
- Lines 47–74 define the `paintingRobotTask` lambda. Lines 56–70 execute until line 56 determines that the task should stop because the assembly line is shutting down. Lines 57–65 simulate the painting work. When painting finishes, the calling `jthread` reaches the `paintingDone` `barrier` and calls its `arrive_and_wait` function (line 69). This decrements the `barrier`'s internal count. If the count is not `0`, the calling `jthread` **blocks** here. If the count is `0`, the `barrier`**'s completion function executes**, moving the next car into position or shutting down the assembly line. When the completion function finishes executing, the `barrier` resets its internal count and **unblocks the waiting** `jthread`s so they can continue executing.
- To begin the simulation, line 76 in `main` moves the first car into position, and lines 80–81 launch two `jthread`s representing the left and right painting robots.

Another mutual-exclusion mechanism is the **semaphore**, as described by Dijkstra in his seminal paper on cooperating sequential processes.[107,108,109] A semaphore contains an integer value representing the maximum number of concurrent threads that can access a shared resource, such as **shared mutable data**. Once initialized, that integer can be accessed and altered by only two operations, **P** and **V**. These are short for the Dutch words *proberen*, meaning "to test," and *verhogen*, meaning "to increase."[110] A thread calls the *P* operation (also called the **wait operation**) when it wants to **enter a critical section** and calls the *V* operation (also called the **signal operation**) when it wants to **exit a critical section**. Once the maximum number of threads are operating in the **critical section**, other threads trying to enter must wait. *P* and *V* are abstractions that encapsulate the details of mutual exclusion implementations. They can support any number of cooperating threads. C++'s semaphore classes call these operations `acquire` and `release`. For a nice discussion of semaphore fundamentals, check out Allen B. Downey's *The Little Book of Semaphores*.[111]

107. E. W. Dijkstra, "Cooperating Sequential Processes," Technological University, Eindhoven, Netherlands, 1965, reprinted in F. Genuys, ed., *Programming Languages*, pp. 43–112. New York: Academic Press, 1968.

108. Harvey Deitel, Paul Deitel and David Choffnes, Chapter 5, "Asynchronous Concurrent Execution." *Operating Systems, 3/e*, pp. 227–233. Upper Saddle River, NJ: Prentice Hall, 2004.

109. "Semaphore (Programming)." Accessed February 7, 2022. https://en.wikipedia.org/wiki/Semaphore_(programming).

110. "Semaphore (Programming)—Operation Names." Accessed February 7, 2022. https://en.wikipedia.org/wiki/Semaphore_(programming)#Operation_names.

111. Allen B. Downey, *The Little Book of Semaphores* (Version 2.2.1), Section "3.6.4 Barrier Solution," 2016. Green Tea Press. https://greenteapress.com/semaphores/LittleBookOfSemaphores.pdf. License: http://creativecommons.org/licenses/by-nc-sa/4.0. Thanks to one of our reviewers, Anthony Williams, for pointing us to Downey's work.

20 C++20's `<semaphore>` header contains features for implementing **counting semaphores** and **binary semaphores**:

- A `std::counting_semaphore` implements the semaphore concept, which is lower level than `std::latch` and `std::barrier`, but still higher level than `mutex`es, locks, `condition_variable`s and atomics.[112,113] A `counting_semaphore` can allow multiple threads to access a shared resource. Its constructor initializes its internal integer counter. When a thread **acquires the semaphore**, the internal counter decrements by one. If the counter reaches zero, a thread attempting to **acquire the semaphore** will block until the count increases to indicate that the shared resource is available. When a thread **releases the semaphore**, the internal counter increments by one (by default) and threads waiting to **acquire the semaphore** unblock.

112. C++ Standard, "Thread Support Library—Semaphore." Accessed February 7, 2022. https://tim-song-cpp.github.io/cppwp/n4861/thread.sema.

113. "`std::counting_semaphore`, `std::binary_semaphore`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/counting_semaphore.

- A `std::binary_semaphore` is simply a `counting_semaphore` with a count of `1` and can be used like a `std::mutex`.[114]

114. "`std::counting_semaphore`, `std::binary_semaphore`." Accessed February 7, 2022. https://en.cppreference.com/w/cpp/thread/counting_semaphore.

20 Producer–Consumer Using C++20 `std::binary_semaphore`s

Perf The C++ standard says **semaphores "are widely used to implement other synchronization primitives and, whenever both are applicable, can be more efficient than condition variables."**[115] Figure 17.16 reimplements class `SynchronizedBuffer` from Section 17.6 using **`binary_semaphore`s** for **mutual exclusion**. This gives you a neat, simple, higher-level way to replace lower-level `mutex` code. We reuse Fig. 17.8's `main` function, so we do not repeat it here.

115. C++ Standard, "32.7 Semaphore." Accessed February 7, 2022. https://timsong-cpp.github.io/cppwp/n4861/thread.sema.

```
1  // Fig. 17.16: SynchronizedBuffer.h
2  // SynchronizedBuffer using two binary_semaphores to
3  // maintain synchronized access to a shared mutable int.
4  #pragma once
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <semaphore>
8  #include <string>
9
10 using namespace std::string_literals;
11
12 class SynchronizedBuffer {
13 public:
14    // place value into m_buffer
15    void put(int value) {
16       // acquire m_produce semaphore to be able to write to m_buffer
17       m_produce.acquire(); // blocks if it's not the producer's turn
18
19       m_buffer = value; // write to m_buffer
20       m_occupied = true;
21
22       std::cout << fmt::format("{:<40}{}\t\t{}\n",
23          "Producer writes "s + std::to_string(value),
24          m_buffer, m_occupied);
25
26       m_consume.release(); // allow consumer to read
27    }
28
29    // return value from m_buffer
30    int get() {
31       int value; // will store the value returned by get
32
33       // acquire m_consume semaphore to be able to read from m_buffer
34       m_consume.acquire();// blocks if it's not the consumer's turn
35
36       value = m_buffer; // read from m_buffer
37       m_occupied = false;
38
39       std::cout << fmt::format("{:<40}{}\t\t{}\n",
40          "Consumer reads "s + std::to_string(m_buffer),
41          m_buffer, m_occupied);
42
43       m_produce.release(); // allow producer to write
```

```
44          return value;
45       }
46    private:
47       std::binary_semaphore m_produce{1}; // producer can produce
48       std::binary_semaphore m_consume{0}; // consumer can't consume
49       bool m_occupied{false};
50       int m_buffer{-1};// shared by producer and consumer threads
51    };
```

```
Operation                       Buffer              Occupied
---------                       ------              --------

Producer writes 1               1                   true
Consumer reads 1                1                   false
Producer writes 2               2                   true
Consumer reads 2                2                   false
Producer writes 3               3                   true
Consumer reads 3                3                   false
Producer writes 4               4                   true
Consumer reads 4                4                   false
Producer writes 5               5                   true
Consumer reads 5                5                   false
Producer writes 6               6                   true
Consumer reads 6                6                   false
Producer writes 7               7                   true
Consumer reads 7                7                   false
Producer writes 8               8                   true
Consumer reads 8                8                   false
Producer writes 9               9                   true
Consumer reads 9                9                   false
Producer writes 10              10                  true
Producer done producing
Terminating Producer
Consumer reads 10               10                  false

Consumer read values totaling 55
Terminating Consumer
```

Fig. 17.16 `SynchronizedBuffer` using two `binary_semaphores` to maintain synchronized access to a shared mutable `int`.

Figure 17.16 uses two **`std::binary_semaphores`** to coordinate the producer and consumer threads:

- `m_produce` (line 47) is initialized with the count `1`, indicating that **the producer initially can produce** a value because `m_buffer` is empty, and
- `m_consume` (line 48) is initialized with the count `0`, indicating that **the consumer initially cannot consume** a value because `m_buffer` is empty.

This example uses `m_occupied` (line 49) for output purposes only—it does not play a role in this example's thread synchronization.

`SynchronizedBuffer` Member Function `put`

`SynchronizedBuffer`'s logic is simplified with **`std::binary_semaphores`**. When the producer thread calls **`put`** (defined at lines 15–27), line 17 calls **`m_produce`'s `acquire` member function**. If `m_produce`'s count is `0`, the producer thread will be blocked until the count is 1. If the count is `1`, the producer thread **acquires the semaphore**, decreasing its count to 0, and writes a new value into `m_buffer`. The producer cannot produce again until `m_produce`'s count is `1`, which will occur only when the consumer consumes the buffer's value.

When the producer is done updating the buffer, line 26 calls `m_consume`'s `release` member function to increment that semaphore's count to `1`. If a consumer thread is blocked waiting to **acquire** `m_consume`, it unblocks so it can proceed. Otherwise, when a consumer tries to call `get`, it can immediately **acquire** `m_consume` and proceed.

`SynchronizedBuffer` Member Function `get`

When the consumer thread calls `get` (defined at lines 30–45), line 34 calls `m_consume`'s `acquire` **member function**. If `m_consume`'s count is `0`, the consumer thread blocks until the count is `1`. If the count is `1`, the consumer thread **acquires** `m_consume`, decreasing its count to `0`, then reads `m_buffer`'s current value. The consumer cannot consume again until `m_consume`'s count is `1`, which will occur only when the producer writes the next value into the buffer.

When the consumer is done reading from the buffer, line 43 calls `m_produce`'s `release` **member function** to increment that semaphore's count to `1`. If a producer thread is blocked waiting to **acquire** `m_produce`, it unblocks so it can proceed. Otherwise, when a producer tries to call `put`, it can immediately **acquire** `m_produce` and proceed.

17.15 C++23: A Look to the Future of C++ Concurrency

Many new concurrency features are being considered for future C++ versions. Here, we overview several of them and provide links to where you can learn more.[116, 117]

116. "C++23." Accessed February 7, 2022. https://en.wikipedia.org/wiki/C%2B%2B23.
117. Ville Voutilainen, "To Boldly Suggest an Overall Plan for C++23," November 25, 2019. Accessed February 7, 2022. https://wg21.link/p0592.

17.15.1 Parallel Ranges Algorithms

20 23 The C++ Standard Committee is working on parallelized versions of C++20's `std::ranges` **algorithms**. The proposal, "A Plan for C++23 Ranges," indicates that there are implementation issues tied to executors.[118]

118. Barry Revzin, Conor Hoekstra and Tim Song, "A Plan for C++23 Ranges," October 14, 2020. Accessed February 7, 2022. https://wg21.link/p2214.

17.15.2 Concurrent Containers

Perf 23 You've seen various **non-concurrent containers** throughout this book, and in this chapter, you studied a `CircularBuffer` class in which we implemented concurrent access via `std::mutex`, `std::unique_lock` and `std::condition_variable`. Generally, rather than creating your own concurrent containers, as we did, it's good practice to use preexisting ones that manage synchronization for you—such as concurrent queues and concurrent maps (also called concurrent hash tables or concurrent dictionaries). These are written by experts, have been thoroughly tested and debugged, operate efficiently and help you avoid common traps and pitfalls. Such containers might be included in the C++23 standard library. For now, you'll need to use third-party libraries, such as the containers in the **Google Concurrency Library (GCL)**[119] or the **Microsoft Parallel Patterns Library**.[120] For more information on concurrent queues, see the "C++ Concurrent Queues" proposal.[121] The proposal indicates that the **concurrent queue reference implementations** are

provided by the GCL's `buffer_queue` and `lock_free_buffer_queue` classes. For more information on concurrent maps, see the C++ Standards Committee proposals:
119. Alasdair Mackintosh, "Google Concurrency Library (GCL)." Accessed February 7, 2022. https://github.com/alasdairmackintosh/google-concurrency-library.
120. "Parallel Patterns Library (PPL)—Parallel Containers and Objects." Accessed February 7, 2022. https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-containers-and-objects.
121. Lawrence Crowl and Chris Mysen, "C++ Concurrent Queues." Accessed February 7, 2022. https://wg21.link/p0260.

- "Concurrent Associative Data Structure with Unsynchronized View"[122] and
- "Concurrent Map Customization Options (SG1 Version)"[123]

and the **concurrent map reference implementation** at

```
https://github.com/BlazingPhoenix/concurrent-hash-map
```

17.15.3 Other Concurrency-Related Proposals

The following additional concurrency-related C++ Standards Committee proposals are being considered for C++23 and beyond:

- 23 The **"Hazard Pointers"**[124] and **"Concurrent Data Structures: Read-Copy-Update"**[125] proposals introduce features for safely reclaiming resources shared among threads.
- The **"apply() for `synchronized_value<T>`"**[126] proposal enhances an earlier proposal that introduced `synchronized_value<T>`, which automatically uses a `mutex` to synchronize concurrent access to an object of type `T`. The proposed `apply` function would receive as arguments a function and one or more `synchronized_value<T>` objects. It would then call its function argument on each `synchronized_value<T>`, using the associated `mutex` to synchronize access from concurrent threads.

122. Sergey Murylev, Anton Malakhov and Antony Polukhin, "Concurrent Associative Data Structure with Unsynchronized View," June 13, 2019. Accessed February 7, 2022. http://wg21.link/p0652.
123. David Goldblatt. "Concurrent Map Customization Options (Sg1 Version)," June 16, 2019. Accessed February 7, 2022. https://wg21.link/P1761.
124. Maged M. Michael, Michael Wong, Paul McKenney, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, Daisy S. Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Mathias Stearn and Jens Maurer, "Hazard Pointers," April 9, 2021. Accessed February 7, 2022. https://wg21.link/p1121.
125. Paul McKenney, Michael Wong, Maged M. Michael, Geoffrey Romer, Andrew Hunter, Arthur O'Dwyer, Daisy Hollman, JF Bastien, Hans Boehm, David Goldblatt, Frank Birbacher, Erik Rigtorp, Tomasz Kaminski and Jens Maurer, "Concurrent Data Structures: Read-Copy-Update," May 14, 2021. Accessed February 7, 2022. https://wg21.link/p1122.
126. Anthony Williams, "`apply()` for `synchronized_value<T>`," March 2, 2017. Accessed February 7, 2022. https://wg21.link/p0290.

17.16 Wrap-Up

This chapter presented modern standardized C++ concurrency features for enhancing application performance on multi-core systems. We compared sequential, concurrent and parallel execution, and presented a sample thread-life-cycle diagram. We discussed why concurrent programming is

complex and error-prone. **Generally, developers should prefer the simpler, more convenient, high-level, prebuilt concurrency capabilities that we emphasized in this chapter**.

We showed that C++17's high-level parallel algorithms automatically parallelize tasks for better performance. We used the `<chrono>` header's timing capabilities to profile sequential and parallel algorithm performance on multi-core systems and showed the parallel `std::sort` algorithm's significant performance improvement for big enough data sets (when called with the `std::execution::par` execution policy) over its sequential counterpart. We also compared the `std::tranform` algorithm's performance using the `std::execution::par` (parallel) and `std::execution::unseq` (vectorized) execution policies.

We executed tasks in separate threads via C++20's `std::jthread` class, which the C++ Core Guidelines recommend using in preference to C++11's `std::thread` class. We used `jthread`'s integrated cooperative-cancellation support to enable threads to terminate gracefully so that they can complete critical work and correctly release resources.

We introduced producer–consumer relationships, which are popular in concurrent programming, and demonstrated the problems with a producer thread and a consumer thread simultaneously accessing shared mutable data without synchronization. We corrected those problems by synchronizing access to shared mutable data using low-level C++11 concurrency primitives `std::mutex`, `std::condition_variable` and `std::unique_lock`. Next, we used these primitives to implement a synchronized circular buffer to minimize producer–consumer waiting and increase performance.

We used C++11's `std::async` and `std::future` to implicitly create threads, execute tasks asynchronously and communicate their results back to the thread that called `async`. We discussed how `async` uses a `std::promise` "under the hood" for inter-thread communication to return a task's result, or an exception, to the thread that called `async`.

We introduced C++11's atomic types, which enable threads to **share mutable data conveniently without explicit synchronization and locking**. We overviewed several C++20 atomics enhancements and mentioned that the C++ standard library implementations for Visual C++, `g++` and `clang++` use atomics to implement the higher-level C++20 primitives `std::latch`, `std::barrier` and semaphores.

We demonstrated the `std::latch` and `std::barrier` thread-coordination primitives, showing that they do not require `mutex`es and locks to work correctly. Next, we used C++20 semaphores without `mutex`es, locks and condition variables to synchronize access to shared mutable data.

Finally, we mentioned several high-level concurrency capabilities being considered for C++23 and later versions, including concurrent containers and parallel versions of the range-based algorithms. **Most programmers should prefer prebuilt concurrent containers, such as concurrent queues and maps, that encapsulate synchronization. These help avoid the common traps and pitfalls of using low-level synchronization primitives.** Such concurrent containers are not yet part of the C++ standard library, so you'll need to use existing third-party libraries, such as the Google Concurrency Library and the Microsoft Parallel Patterns Library.

In the next chapter, we'll present a detailed treatment of the concurrency feature coroutines—the last of C++20's "big four" features (which also include ranges, concepts and modules). You'll use a high-level, library-based approach to conveniently create coroutines, enabling sophisticated concurrent programming with a simple sequential-like coding style.

18. C++20 Coroutines

Objectives

In this chapter, you'll:

- Understand what coroutines are and how they're used.
- Use keyword `co_yield` to suspend a generator coroutine and return a result.
- Use the `co_await` operator to suspend a coroutine while it waits for a result to become available.
- Use a `co_return` statement to terminate a coroutine and return its result, or simply control, to its caller.
- Use the open-source `generator` library to simplify creating a generator coroutine with `co_yield`.
- Use the open-source `concurrencpp` library to simplify creating coroutines with `co_await` and `co_return`.
- Become aware of coroutine capabilities being contemplated for future C++ versions.

Outline

18.1 Introduction
18.2 Coroutine Support Libraries
18.3 Installing the `concurrencpp` and `generator` Libraries
18.4 Creating a Generator Coroutine with `co_yield` and the `generator` Library
18.5 Launching Tasks with `concurrencpp`
18.6 Creating a Coroutine with `co_await` and `co_return`
18.7 Low-Level Coroutines Concepts
18.8 C++23 Coroutines Enhancements
18.9 Wrap-Up

18.1 Introduction

20 SE When a program contains a long-running task, it's common for a function that you call **synchronously**—that is, performing tasks one after another—to launch the long-running task **asynchronously**. Typically, the program would provide that asynchronous task with a **callback function** (a function, a lambda or a function object) to call when the task completes. This coding style is simplified with **C++20 coroutines**—the last of C++20's "big four" features (ranges, concepts, modules and coroutines) we cover in this book.

A **coroutine**[1] is a function that can **suspend execution and be resumed later**. The mechanisms that

1. The term "coroutine" was coined by Melvin Conway in 1958. "Coroutine." Accessed February 12, 2022. https://en.wikipedia.org/wiki/Coroutine.

- suspend a coroutine and return control to its caller, and
- continue a suspended coroutine's execution later

are **handled entirely by code that's written for you by the compiler**. You'll see that a function containing any of the keywords `co_await`, `co_yield` or `co_return` is a **coroutine**.

SE **Coroutines enable you to do concurrent programming with a simple sequential-like coding style.** This capability requires sophisticated infrastructure. You can write the infrastructure

yourself, but doing so is complex, tedious and error-prone. Instead, most experts agree **programmers should use high-level coroutine support libraries**, which is the approach we demonstrate. The open-source community has created several experimental libraries for developing coroutines quickly and conveniently. Section 18.2 lists several and provides the rationale for the two we use in our coroutines examples.

Coroutine Use Cases

Some coroutines use cases[2] include:

2. Geoffrey Romer, Gor Nishanov, Lewis Baker and Mihail Mihailov, "Coroutines: Use-Cases and Trade-Offs," February 19, 2019. Accessed February 12, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1493r0.pdf.

- web servers handling requests, enabling a function's execution to span multiple animation frames in game programming, consuming data once it becomes available (such as the results of long-running calculations; Section 18.5), data coming from devices in the Internet of Things (IoT) or downloads of large files;[3]

3. "What Are Use Cases for Coroutines?" Accessed February 12, 2022. https://stackoverflow.com/questions/303760/what-are-use-cases-for-coroutines.

- lazily computed sequences (known as **generators**; Section 18.4) that produce one value at a time on demand;[4]

4. "Coroutines," Accessed February 12, 2022. https://en.cppreference.com/w/cpp/language/coroutines.

- event-driven coding without callback functions[5]—for example, simulations, user interfaces, servers, games, non-blocking I/O;[6]

5. David Mazières, "My Tutorial and Take on C++20 Coroutines," February 2021. Accessed February 12, 2022. https://www.scs.stanford.edu/~dm/blog/c++-coroutines.html.

6. Techmunching, "Coroutines and Their Introduction in C++," May 30, 2020. Accessed February 12, 2022. https://techmunching.com/coroutines-and-their-introduction-in-c/.

- cooperative multitasking;[7, 8]

7. Rainer Grimm, "C++20: More Details to Coroutines," March 27, 2020. Accessed February 12, 2022. http://modernescpp.com/index.php/component/content/article/54-blog/c-20/488-c-20-coroutines-more-details.

8. Bobby Priambodo, Cooperative vs. Preemptive: a Quest to Maximize Concurrency Power," September 3, 2019. Accessed February 12, 2022. https://medium.com/traveloka-engineering/cooperative-vs-preemptive-a-quest-to-maximize-concurrency-power-3b10c5a920fe.

- structured concurrency;[9]

9. Lewis Baker, "Structured Concurrency: Writing Safer Concurrent Code with Coroutines and Algorithms," October 14, 2019. Accessed February 12, 2022. https://www.youtube.com/watch?v=1Wy5sq3s2rg.

- reactive streams programming;[10] and

10. Jeff Thomas, "Exploring Coroutines," April 7, 2021. Accessed February 12, 2022. https://blog.coffeetocode.com/2021/04/exploring-coroutines/.

- implementing state machines.[11]

11. Steve Downey, "Converting a State Machine to a C++ 20 Coroutine," June 29, 2021. Accessed February 12, 2022. https://www.youtube.com/watch?v=Z8jHi9Cs6Ug.

18.2 Coroutine Support Libraries

20 Coroutines require various supporting classes with numerous member functions and nested types. Creating these yourself is cumbersome, complex and error-prone. The C++20 standard library

includes only the low-level primitives that library writers need to build coroutines support libraries. Lewis Baker, who has worked on several such libraries, said that these primitives "can be thought of as a low-level assembly-language for coroutines."[12] Most developers will prefer to work with a coroutine support library. Standard library coroutine support is expected in C++23.[13, 14]

12. Lewis Baker, "C++ Coroutines: Understanding Operator `co_await`," November 17, 2017. Accessed February 12, 2022. https://lewissbaker.github.io/2017/11/17/understanding-operator-co-await.
13. Ville Voutilainen, "To Boldly Suggest an Overall Plan for C++23," November 25, 2019. Accessed February 12, 2022. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0592r4.html.
14. "C++23." Accessed February 12, 2022. https://en.wikipedia.org/wiki/C%2B%2B23.

23 In the interim, the open-source community has created several non-standard experimental coroutine support libraries. Here are the ones we considered for this presentation:

- Lewis Baker's `cppcoro`[15] is frequently mentioned in many books, videos and blog posts but is no longer supported.[16]

15. Lewis Baker, "`cppcoro`." Accessed February 12, 2022. https://github.com/lewissbaker/cppcoro.
16. In a September 18, 2021 email interaction, Mr. Baker indicated that `cppcoro` is no longer supported. He now works on Facebook's experimental `folly::coro` and `libunifex` libraries.

- Facebook's `folly::coro` is a subset of its popular and large `folly` open-source C++ utilities library.[17] Lewis Baker is part of the Facebook team responsible for `folly::coro`. This library's documentation indicates that `folly::coro` is experimental. Unfortunately, `folly::coro` cannot be installed independently of `folly`.

17. "facebook/folly." Accessed February 12, 2022. https://github.com/facebook/folly.

- David Haim's `concurrencpp`[18] is actively maintained. This library enables you to conveniently develop coroutines using `co_await` and `co_return`. At the time of this writing, `concurrencpp` did not have `co_yield` support, but David added it in `concurrencpp` version 0.1.4.[19]

18. David Haim, "`concurrencpp`." Accessed February 12, 2022. https://github.com/David-Haim/concurrencpp.
19. Per a November 14, 2021 email interaction we had with David Haim.

- Perf Sy Brand's `generator`[20] is a header-only library for developing coroutines that return one value at a time via `co_yield`. A generator coroutine produces values **on demand**, known as **lazy evaluation**. This is in contrast to **greedy evaluation**—for example, the `std::ranges::generate` algorithm (Section 14.4.1) **immediately** generates values and places them into a range, such as a `vector`. For large numbers of items, creating a range can take substantial memory and time. If a program does not need all the values at once, generators can reduce your program's memory consumption and improve performance. Generators also can define infinite sequences, such as the Fibonacci sequence (Section 18.4) or prime numbers.[21]

20. Sy Brand (C++ Developer Advocate, Microsoft), "`generator`." Accessed February 12, 2022. https://github.com/TartanLlama/generator.
21. Visual C++ provides a `std::experimental::generator` implementation in its `<experimental/generator>` header.

For our coroutines presentation, we use `concurrencpp` and `generator`.[22] `concurrencpp` installs easily, is actively maintained, and has clear documentation with many code examples. It provides **high-level concurrency features**, including

22. Thank you to Anthony Williams (https://www.linkedin.com/in/anthonyajwilliams)—author of *C++ Concurrency in Action, Second Edition* (https://www.manning.com/books/c-plus-plus-concurrency-in-action-second-edition)—for sharing his thoughts with us as we were deciding how to approach this coroutines section.

- **tasks** for executing functions **asynchronously**,
- **executors** for **scheduling tasks**, possibly executing them in separate threads,
- **timers** for **performing tasks in the future** and
- various **utility functions**.

23 20 In Section 18.4's example, we use the `generator` library to implement a Fibonacci generator coroutine that generates the next Fibonacci value in sequence on demand and returns it to the caller via `co_yield`. Section 18.5's example introduces `concurrencpp` **tasks** and **executors**—standardized versions of each are expected in C++23. Section 18.6's example uses these to implement a coroutine demonstrating `co_await` and `co_return`.

18.3 Installing the `concurrencpp` and `generator` Libraries

`concurrencpp` Library

To install `concurrencpp`, follow the instructions for your platform at

```
https://github.com/David-Haim/concurrencpp#building-installing-and-testing
```

If you're using Visual C++, perform the following additional steps:

1. Open `concurrencpp.sln` from the library's `concurrencpp\build\lib` and build the solution to generate the library files.
2. In your Visual Studio solution that will use `concurrencpp`, follow the instructions presented in Section 3.12 to add the `concurrencpp\include` folder to the header include path.
3. Next, select **File > Add > Existing Project...**, navigate to the `concurrencpp` library's `concurrencpp\build\lib` folder and add `concurrencpp.vcxproj` to your solution.
4. Finally, right-click your project, select **Add > Reference...**, then in the **Add Reference** dialog, check the `concurrencpp` project and click **OK**.

`generator` Library

You can simply download the header-only `generator` library from

```
https://github.com/TartanLlama/generator
```

and include it in your project. If you prefer, you can clone the GitHub repository, then add the library's `include` folder to your compiler's header include path.

18.4 Creating a Generator Coroutine with `co_yield` and the `generator` Library

23 A **generator** is a coroutine that produces values on demand. When you call a generator, it uses a **`co_yield` expression** **to suspend its execution and return the next generated value** to its caller. Generator support is expected to be part of the C++23 standard library. For this example, we'll use the `generator` library.[23] Its **`tl::generator` class template** enables you to specify the return type of a generator coroutine. It provides the mechanisms that enable `co_yield` to return values to a generator coroutine's caller.

23. At the time of this writing, the `generator` library does not work with `clang++`.

Figure 18.1 defines a generator coroutine that produces the Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

which begins with 0 and 1 and has the property that each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers.

```
1  // fig18_01.cpp
2  // Creating a generator coroutine with co_yield.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <sstream>
6  #include <thread>
7  #include <tl/generator.hpp>
8
9  // get current thread's ID as a string
10 std::string id() {
11    std::ostringstream out;
12    out << std::this_thread::get_id();
13    return out.str();
14 }
15
16 // coroutine that repeatedly yields the next Fibonacci value in sequence
17 tl::generator<int> fibonacciGenerator(int limit) {
18    std::cout << fmt::format(
19       "Thread {}: fibonacciGenerator started executing\n", id());
20
21    int value1{0}; // Fibonacci(0)
22    int value2{1}; // Fibonacci(1)
23
24    for (int i{0}; i < limit; ++i) {
25       co_yield value1; // yield current value of value1
26
27       // update value1 and value2 for next iteration
28       int temp{value1 + value2};
29       value1 = value2;
30       value2 = temp;
31    }
32
33    std::cout << fmt::format(
34       "Thread {}: fibonacciGenerator finished executing\n", id());
35 }
36
37 int main() {
38    std::cout << fmt::format("Thread {}: main begins\n", id());
39
40    // display first 10 Fibonacci values
41    for (int i{0}; auto value : fibonacciGenerator(10)) {
42       std::cout << fmt::format("Fibonacci({}) is {}\n", i++, value);
43    }
44
45    std::cout << fmt::format("Thread {}: main ends\n", id());
```

```
46    }
Thread 7316: main begins
Thread 7316: fibonacciGenerator started executing
Fibonacci(0) is 0
Fibonacci(1) is 1
Fibonacci(2) is 1

Fibonacci(3) is 2
Fibonacci(4) is 3
Fibonacci(5) is 5
Fibonacci(6) is 8
Fibonacci(7) is 13
Fibonacci(8) is 21
Fibonacci(9) is 34
Thread 7316: fibonacciGenerator finished executing
Thread 7316: main ends
```

Fig. 18.1 Creating a generator coroutine with `co_yield`.

`id` Function for Converting a Thread ID to a `std::string`

The `id` function (lines 10–14) uses a `std::thread::id` object's overloaded `operator<<` to convert a thread ID to a `std::string`. We'll use this to show that **`main` and `fibonacci-Generator` execute in the same thread**.

`fibonacciGenerator` Coroutine

Lines 17–35 define the `fibonacciGenerator` coroutine. The function's `limit` parameter determines the number of Fibonacci values to produce, starting with Fibonacci(0). Lines 18–19 display the thread ID in which `fibonacciGenerator` is executing. In the program's output, you can see that the thread ID is the same as the one displayed by `main`, showing that **the coroutine executes in the same thread as `main`**. Lines 21 and 22 define variables `value1` and `value2`. Initially, these variables store the first two values in the Fibonacci sequence. **When a program requests a new value from the generator, `co_yield`** (line 25) **suspends the coroutine's execution and immediately returns the next Fibonacci value**. When the code subsequently asks for the next value, the coroutine **resumes** and lines 28–30 update `value1` and `value2`. Then, the next iteration of the loop **`co_yield`**s the next Fibonacci value, once again **suspending** the coroutine and returning a value to the caller. This continues until the last Fibonacci value is produced. Then, lines 33–34 again display the thread ID in which `fibonacciGenerator` is executing to show that it's still the same thread as `main`.

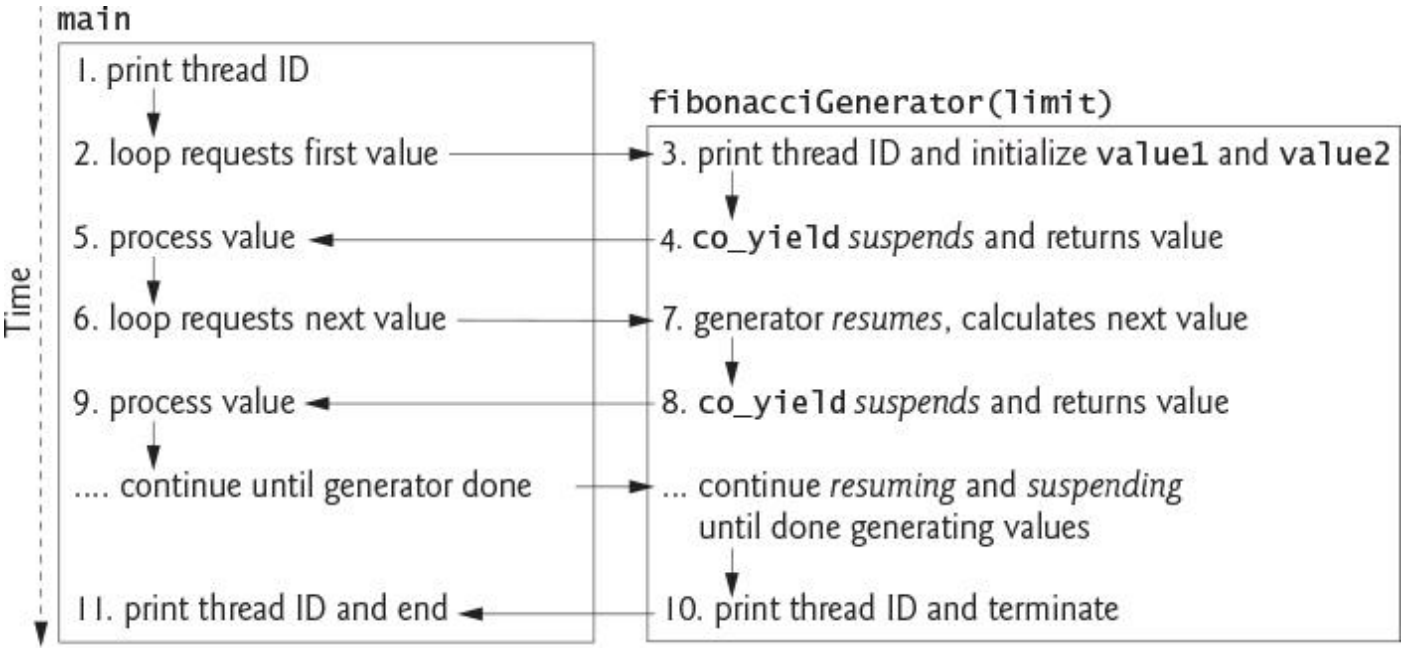
`main` Function

Line 38 displays `main`'s thread ID, so we can **confirm that `main` and `fibonacciGenerator` execute in the same thread**. Lines 41–43 request the Fibonacci(0) through Fibonacci(9) values. The call `fibonacciGenerator(10)` returns a **`tl::generator<int>`**, which provides iterators, so you can use it in a range-based `for` loop. When each iteration of the loop requests the next value, the `fibonacciGenerator` coroutine **resumes** execution to produce the next value, then **suspends and returns it**. Finally, line 45 displays `main`'s thread ID and shows that `main` ends. The thread IDs in the sample output confirm that **`fibonacciGenerator` performs its work in `main`'s thread**.

SE **Calling a coroutine does not automatically create a new thread for you. If any threads are required, the coroutine must create them**, as we'll do in Sections 18.5 and 18.6. You can create and manage those threads using the techniques shown in Chapter 17, or you can let libraries like **`concurrencpp`** create and manage them for you.

Diagram Showing the Flow of Control for a Generator Coroutine

The following diagram shows the basic flow of control in this program—the numbers in this description correspond to the steps in the diagram:


main
1. print thread ID
2. loop requests first value
5. process value
6. loop requests next value
9. process value
.... continue until generator done
11. print thread ID and end
fibonacciGenerator(limit)
3. print thread ID and initialize value1 and value2
4. co_yield suspends and returns value
7. generator resumes, calculates next value
8. co_yield suspends and returns value
... continue resuming and suspending until done generating values
10. print thread ID and terminate
Time

1. `main` prints its thread ID.
2. The loop in `main` requests the first value from `fibonacciGenerator`.
3. `fibonacciGenerator` prints its thread ID and initializes its local variables.
4. `fibonacciGenerator` `co_yield`s a value, which suspends the coroutine's execution and returns the value, and control, to `main`.
5. The loop in `main` processes the value.
6. The loop in `main` requests the next value from `fibonacciGenerator`.
7. `fibonacciGenerator` resumes execution and calculates the next value.
8. `fibonacciGenerator` `co_yield`s a value. Again, this suspends the coroutine's execution and returns the value, and control, to `main`.
9. The loop in `main` processes the value.
10. Steps 6–9 continue until `fibonacciGenerator` finishes producing values.

Then, `fibonacciGenerator` prints its thread ID, terminates and returns control to `main`.

11. The loop in `main` terminates, then `main` prints its thread ID and terminates.

Coroutines Are Stackless

Perf The compiler manages the mechanisms that enable coroutines to suspend and resume. It creates the **coroutine state**,[24] which contains the information required to resume a coroutine. This state is dynamically allocated on the heap rather than the stack, so coroutines are said to be **stackless**.[25] If the compiler determines that the coroutine's lifetime is entirely within that of its caller, it can eliminate the heap allocation overhead.[26] Marc Gregoire points out that "memory usage for stackless coroutines is minimal, allowing for millions or even billions of coroutines to be running concurrently."[27]

24. C++ Standard, "Coroutine Definitions," Accessed February 12, 2022. https://timsongcpp.github.io/cppwp/n4861/dcl.fct.def.coroutine#9.
25. Varun Ramesh Blog, "Stackless vs. Stackful Coroutines," August 18, 2017. Accessed February 12, 2022. https://blog.varunramesh.net/posts/stackless-vs-stackful-coroutines/.

26. “Coroutines—Heap Allocation.” Accessed February 12, 2022. https://en.cppreference.com/w/cpp/language/coroutines#Heap_allocation.
27. Marc Gregoire, *Professional C++, Fifth Edition*, p. 963. 2021. Indianapolis, IN: John Wiley & Sons, 2021.

18.5 Launching Tasks with `concurrencpp`

Before we use `concurrencpp` to implement a coroutine, let’s use it to **schedule tasks that execute in separate threads**. As with standard library threads, each task executes a function, a lambda or a function object. Figure 18.2 uses four `concurrencpp` components:

- A `concurrencpp::runtime` manages your `concurrencpp` interactions. It provides access to various executors for scheduling tasks. It also ensures that the executors shut down scheduled tasks properly when the `runtime` is destroyed. You must create a local `runtime` object, typically at the beginning of `main` (line 35). When `main` ends, the local `runtime` object goes out of scope. Its destructor shuts down the executors and any remaining scheduled tasks that have not finished executing.
- Perf 23 A `concurrencpp::thread_pool_executor` (one of several executor types) schedules tasks to execute. It creates and manages a group of threads called a **thread pool**[28] and assigns tasks to the pool’s threads to execute. This executor can **reuse existing threads** in the pool to **eliminate the overhead of creating a new thread for each task**. It also can **optimize the number of threads** to ensure the processor stays busy without creating so many threads that the application runs out of resources. Standard executors are expected in C++23, but libraries like `concurrencpp`, `libunifex`[29] and `folly::coro` offer them now.

28. “Thread pool,” Accessed February 12, 2022. https://en.wikipedia.org/wiki/Thread_pool.
29. “`libunifex`.” Accessed February 12, 2022. https://github.com/facebookexperimental/libunifex.

- A `concurrencpp::task` represents a task to execute. `concurrencpp` **executors** create these when you schedule tasks.
- A `concurrencpp::result` enables you to access a concurrent task’s result or simply wait for the task to complete execution if it does not return a value. When you schedule a `concurrencpp::task`, you receive a `concurrencpp::result`.

```
1  // fig18_02.cpp
2  // Setting up the concurrencpp::runtime and scheduling tasks with it.
3  #include <chrono>
4  #include <concurrencpp/concurrencpp.h>
5  #include <fmt/format.h>
6  #include <iostream>
7  #include <random>
8  #include <sstream>
9  #include <thread>
10 #include <vector>
11
12 // get current thread's ID as a string
13 std::string id() {
14    std::ostringstream out;
15    out << std::this_thread::get_id();
16    return out.str();
17 }
18
19 // Function printTask sleeps for a specified period in milliseconds.
```

```
20    // When it continues executing, it prints its name and completes.
21    void printTask(std::string name, std::chrono::milliseconds sleep) {
22       std::cout << fmt::format(
23          "{} (thread ID: {}) going to sleep for {} ms\n",
24          name, id(), sleep.count());
25
26       // put the calling thread to sleep for sleep milliseconds
27       std::this_thread::sleep_for(sleep);
28
29       std::cout << fmt::format("{} (thread ID: {}) done sleeping\n",
30          name, id());
31    }
32
33    int main() {
34       // set up the concurrencpp runtime for scheduling tasks to execute
35       concurrencpp::runtime runtime;
36
37       std::cout << fmt::format("main's thread ID: {}\n\n", id());
38
39       // set up random number generation for random sleep times
40       std::random_device rd;
41       std::default_random_engine engine{rd()};
42       std::uniform_int_distribution ints{0, 5000};
43
44       // stores the tasks so we can wait for them to complete later;
45       // concurrencpp::result<void> indicates that each task returns void
46       std::vector<concurrencpp::result<void>> results;
47
48       std::cout << "STARTING THREE CONCURRENCPP TASKS\n";
49
50       // schedule three tasks
51       for (int i{1}; i < 4; ++i) {
52          std::chrono::milliseconds sleepTime{ints(engine)};
53          std::string name{fmt::format("Task {}", i)};
54
55          // use a concurrencpp thread_pool_executor to schedule a call
56          // to printTask with name and sleepTime as its arguments
57          results.push_back(runtime.thread_pool_executor()->submit(
58             printTask, name, sleepTime));
59       }
60
61       std::cout << "\nALL TASKS STARTED\n";
62       std::cout << "\nWAITING FOR TASKS TO COMPLETE\n";
63
64       // wait for each task to complete
65       for (auto& result : results) {
66          result.get(); // wait for each task to return its result
67       }
68
69       std::cout << fmt::format("main's thread ID: {}\nMAIN ENDS\n", id());
70    }
```

```
main's thread ID: 20740

STARTING THREE CONCURRENCPP TASKS
```

```
ALL TASKS STARTED

WAITING FOR TASKS TO COMPLETE
Task 3 (thread ID: 18960) going to sleep for 2683 ms
Task 1 (thread ID: 9840) going to sleep for 3856 ms
Task 2 (thread ID: 1700) going to sleep for 904 ms
Task 2 (thread ID: 1700) done sleeping
Task 3 (thread ID: 18960) done sleeping
Task 1 (thread ID: 9840) done sleeping

main's thread ID: 20740
MAIN ENDS
```

Fig. 18.2 Setting up the `concurrencpp::runtime` and scheduling tasks with it.

Function `id`

Function `id` (lines 13–17) creates a `std::string` representation of the current thread's unique ID. We'll use this to show the threads in which `main` and each of our tasks execute.

Function `printTask`

We do not create and manage the threads in this example. **When you schedule tasks with `concurrencpp::thread_pool_executor`, it creates and manages threads for you using its thread pool.** This example schedules calls to function `printTask` (lines 21–31) and shows that they execute on separate threads. Lines 22–24 and 29–30 display the currently executing thread's unique ID to **confirm that `printTask` is called from separate threads**.

Function `main`

`main` is similar to the one in Fig. 17.4, so we'll focus on the `concurrencpp` statements. Line 35 creates the `concurrencpp::runtime` so we can use the library's features. Line 37 displays the thread ID in which `main` is executing.

`concurrencpp` Tasks and Results

Tasks allow developers to be more productive "by allowing them to focus more on business logic and less on low-level concepts like thread management and inter-thread synchronizations."[30] You define your tasks as functions (or lambdas or function objects). When you schedule a task, `concurrencpp` creates a `concurrencpp::task` object for you and returns a `concurrencpp::result` object representing the task's result, which might not be available until sometime in the future. If the task's function returns a result, you access it through the `concurrencpp::result` object, as we'll show in Fig. 18.3. If the function returns `void`, as `printTask` does, you can use the **`concurrencpp::result<void>` object** to wait for the task to complete, similar to joining a thread. Line 46 creates a `vector` of `concurrencpp::result<void>` objects. **We use this to store the `task`s' `result`s, so we can wait for the `task`s to complete later in `main`.**

30. David Haim, "`concurrencpp` overview," Accessed February 12, 2022. https://github.com/David-Haim/concurrencpp#concurrencpp-overview.

`concurrencpp::thread_pool_executor`

Programming languages like Java, Go, Python and Kotlin recommend implementing concurrency via an executor rather than creating and managing threads directly. This example uses a `concurrencpp::thread_pool_executor` (lines 57–58) to schedule each task.

The `runtime` object's `thread_pool_executor` function returns a `shared_ptr`. **All parts of your program**

use `shared_ptr`s to interact with a single object of a given executor type. The executor's `submit` function schedules a task to execute. It receives a variable number of arguments:

- `submit`'s first argument is the function defining the task to execute—in this case, `printTask`.
- `submit` passes its additional arguments to the function in its first argument—in this case, `submit` passes `name` and `sleepTime` to `printTask`.

Each `submit` call creates a `task` object and returns a `result`. Again, `printTask` returns `void`, so `submit` returns type `concurrencpp::result<void>` in this example.

Waiting for the Tasks to Complete Execution

Calling each `result` object's `get` function (lines 65–67) waits for the corresponding task's result—similar to joining a thread. Each task in this example returns `void`, so calling `get` simply causes `main` to wait for the task to complete. If `printTask` returned a value, `get` would return that value.

Summary of `concurrencpp` Executors

The `concurrencpp` documentation provides a list of executors and when to use each,[31] including:

31. "`concurrencpp`—Executors." Accessed February 12, 2022. https://github.com/David-Haim/concurrencpp#executors. Copyright 2020 David Haim.

- `thread_executor`—For each task, a `thread_executor` launches a separate thread that is not reused once it completes. According to the `concurrencpp` documentation, `thread_executor` "is good for long-running tasks, like objects that run a work loop, or long blocking operations."
- `thread_pool_executor` (used in Figs. 18.2 and 18.3)—Recall that this schedules tasks using a pool of threads. "Suitable for short CPU-bound tasks that don't block. Applications are encouraged to use this executor as the default executor for non-blocking tasks."
- `background_executor`—"A thread pool executor with a larger pool of threads. Suitable for launching short blocking tasks like file I/O and DB [database] queries."
- `worker_thread_executor`—"A single thread executor that maintains a single task queue. Suitable when applications want a dedicated thread that executes many related tasks."
- `inline_executor` (discussed momentarily)—"Mainly used to override the behavior of other executors. Enqueuing a task is equivalent to invoking it inline."

Tasks Are Not Required to Run on Separate Threads

A `concurrencpp::inline_executor` schedules tasks on the calling thread. To produce the following output, we replaced the `thread_pool_executor` (Fig. 18.2, line 57) with an `inline_executor`, which executes tasks on the calling thread (in this case, `main`). Every time you run this program with an `inline_executor`, **the tasks execute sequentially in the same thread as `main` and in the order you schedule them**. This sample output shows that all three tasks have the same thread ID as `main`. As we schedule each task, it immediately goes to sleep, then eventually wakes up and completes **before the next task launches**.

```
main's thread ID: 7854

STARTING THREE CONCURRENCPP TASKS
Task 1 (ID: 7854) going to sleep for 2000 ms
Task 1 (ID: 7854) done sleeping
Task 2 (ID: 7854) going to sleep for 4432 ms
Task 2 (ID: 7854) done sleeping
Task 3 (ID: 7854) going to sleep for 3688 ms
Task 3 (ID: 7854) done sleeping
```

```
ALL TASKS STARTED

WAITING FOR TASKS TO COMPLETE

main's thread ID: 7854
MAIN ENDS
```

18.6 Creating a Coroutine with `co_await` and `co_return`

Now, let's use `concurrencpp`, `co_await` and `co_return` to implement a coroutine that performs a potentially long-running task—sorting a 100-million-element `vector` of `int` values.

Overview of This Example

We'll use `concurrencpp`'s `thread_pool_executor` to put two tasks to work in parallel, with each sorting half the `vector`. Once those tasks are complete, we'll use the standard library algorithm `inplace_merge` (Section 14.4.9) to merge the `vector`'s two sorted halves. Figure 18.3 consists of the following functions:

- Function `id` (lines 14–18) converts a unique thread ID to a `string`.
- Coroutine `sortCoroutine` (lines 21–78) uses `concurrencpp` to launch two tasks that each sort half the `vector`.
- Function `main` (lines 80–107) creates the 100-million-element `vector` and calls `sortCoroutine` to sort the `vector`.

We've split this program into pieces for discussion purposes. After the program, we show a sample output.

`#include` Directives and Function `id`

In Fig. 18.3, lines 3–11 include the headers used in this program, and lines 14–18 define function `id`. The `concurrencpp` library (line 3) provides the coroutine support features this program requires.

```
 1  // fig18_03.cpp
 2  // Implementing a coroutine with co_await and co_return.
 3  #include <concurrencpp/concurrencpp.h>
 4  #include <fmt/format.h>
 5  #include <iostream>
 6  #include <memory> // for shared_ptr
 7  #include <random>
 8  #include <sstream>
 9  #include <string>
10  #include <thread>
11  #include <vector>
12
13  // get current thread's ID as a string
14  std::string id() {
15     std::ostringstream out;
16     out << std::this_thread::get_id();
17     return out.str();
18  }
19
```

Fig. 18.3 Implementing a coroutine with `co_await` and `co_return`.

`sortCoroutine` That Sorts Launches Two Tasks

Lines 21–78 define `sortCoroutine`, which receives as arguments

- a `std::shared_ptr<concurrencpp::thread_pool_executor>` used to schedule tasks and
- a `const` reference to a `vector<int>` to sort.

```
20   // coroutine that sorts a vector<int> using two tasks
21   concurrencpp::result<void> sortCoroutine(
22      std::shared_ptr<concurrencpp::thread_pool_executor> executor,
23      std::vector<int>& values) {
24
25      std::cout << fmt::format("Thread {}: sortCoroutine started\n\n", id());
26
27      // lambda that sorts a portion of a vector
28      auto sortTask{
29         [&](auto begin, auto end) {
30            std::cout << fmt::format(
31               "Thread {}: Sorting {} elements\n", id(), end - begin);
32            std::sort(begin, end);
33            std::cout << fmt::format("Thread {}: Finished sorting\n", id());
34         }
35      };
36
37      // stores task results
38      std::vector<concurrencpp::result<void>> results;
39
40      size_t middle{values.size() / 2}; // middle element index
41
42      std::cout << fmt::format(
43         "Thread {}: sortCoroutine starting first half sortTask\n", id());
44
45      // use a concurrencpp thread_pool_executor to schedule
46      // a sortTask call that sorts the first half of values
47      results.push_back(
48         executor->submit(
49            [&]() {sortTask(values.begin(), values.begin() + middle);}
50         )
51      );
52
53      std::cout << fmt::format(
54         "Thread {}: sortCoroutine starting second half sortTask\n", id());
55
56      // use a concurrencpp thread_pool_executor to schedule
57      // a sortTask call that sorts the second half of values
58      results.push_back(
59         executor->submit(
60            [&]() {sortTask(values.begin() + middle, values.end());}
61         )
62      );
63
64      // suspend coroutine while waiting for all sortTasks to complete
65      std::cout << fmt::format("\nThread {}: {}\n", id(),
66         "sortCoroutine co_awaiting sortTask completion");
67      co_await concurrencpp::when_all(
```

```
68          executor, results.begin(), results.end());
69
70       // merge the two sorted sub-vectors
71       std::cout << fmt::format(
72          "\nThread {}: sortCoroutine merging results\n", id());
73       std::inplace_merge(
74          values.begin(), values.begin() + middle, values.end());
75
76       std::cout << fmt::format("Thread {}: sortCoroutine done\n", id());
77       co_return; // terminate coroutine and resume caller
78    }
79
```

The `sortCoroutine` operates as follows:

- Line 25 displays a message containing the thread ID in which `sortCoroutine` is running and indicating that `sortCoroutine` started. **The thread ID confirms that `sortCoroutine` runs in the same thread as `main`.**
- The `sortCoroutine` will launch two `concurrencpp` tasks that execute the lambda `sortTask` (lines 28–35). The lambda receives random-access iterators representing the beginning and end of a common range to sort. Lines 30–31 display a message containing the thread ID in which a given call to `sortTask` executes and the number of elements it is sorting. Line 32 calls the `std::sort` algorithm to sort the specified portion of the `vector`. When the sort completes, line 33 displays a message containing the thread ID and indicating that the task's sort completed.
- The `sortTask` does not return a value, so each `thread_pool_executor`'s `submit` **function** will return a `concurrencpp::result<void>` for each task. We'll store these in the `vector results` (defined at line 38).
- Line 40 determines the `vector`'s middle element, which we'll use to divide the vector in half.
- Lines 42–43 display a message indicating the executing thread's ID and that we're starting the `sortTask` for the first half of the `vector`. Then lines 47–51 use `thread_pool_executor`'s `submit` function to create a task that calls `sortTask` to sort the elements in the range `values.begin()` up to, but not including, the middle `values` element. We store the `result` object returned by `submit` in the `results` `vector`. Lines 53–62 repeat these steps to launch a second task that sorts the `values` elements from the `middle` element through the end of the `vector`.
- Lines 66–66 display a message containing the thread ID in which `sortCoroutine` is running and indicate that we are `co_await`ing the `sortTask` results. Lines 67–68 use `concurrencpp`'s `when_all` function to `co_await` all the `sortTask`' results. We'll discuss this line in more detail momentarily.
- Lines 71–72 display a message containing the thread ID in which `sortCoroutine` is running and indicate that we are merging the results. Then, lines 73–74 use the standard library algorithm `inplace_merge` to merge the `vector`'s two sorted halves.
- 20 Finally, line 76 displays the thread ID of the thread in which `isPrime` is currently executing and indicates that `sortCoroutine` is done. Then, line 77 uses the C++20 `co_return` **statement** to terminate the coroutine and return control to `main`. Because `sortCoroutine` does not return a value, `concurrencpp` returns a `concurrencpp::result<void>` to the coroutine's caller. You'll see that `main` uses this object's `get` **function** to wait for the coroutine to finish executing.

20 Using `when_all` with C++20's `co_await` Expression

23 SE A coroutine utility function we might see in C++23 or later is `when_all`, which enables a program to wait for a set of tasks to complete. Function `when_all` receives as arguments

- an executor that's used to resume the coroutine's execution once all the tasks are complete and

- iterators representing a **common range** of `concurrencpp::result` objects—in this case, all the elements of the `results vector`.

SE Lines 67–68 `co_await` the result of the `when_all` call. A `co_await` expression consists of the `co_await` operator followed by an expression that returns an awaitable entity. Typically, this will be an object of a coroutine-library class. A `concurrencpp::result` satisfies the C++ standard's requirements for the operand of the `co_await` **operator**, as does the object `when_all` returns, which is a `concurrencpp::lazy_result` object containing a `tuple` of all the tasks' results.

Determining Whether to Suspend Coroutine Execution

At this point, the coroutine determines whether to **suspend** its execution. Before suspending, the coroutine calls the `co_await` operand's `await_ready` function to check whether a task's result is available. If so, both `sortTask`s already completed, so `sortCoroutine` continues executing with the next statement in sequence. Otherwise, **the `sortCoroutine` coroutine suspends execution until the asynchronous tasks in `when_all`'s arguments complete execution**. This allows the caller (`main`) to perform other work that does not depend on the results of the asynchronous tasks.

As you'll see in the sample output, if the coroutine suspends, `main` shows that it's executing again by displaying a line of text. Then, `main` waits for `sortCoroutine` to complete and return. However, rather than waiting for `sortCoroutine`, `main` could continue doing other work in the meantime.

Resuming Coroutine Execution

When the `co_await`**ed asynchronous tasks** are complete, the `sortCoroutine` resumes execution and continues with the next statement after the `co_await` **expression**. In this example, `sortCoroutine` merges the sorted first half and sorted second half of the vector, then `co_return`**s** control to `main` so it can continue executing.

`concurrencpp when_any` Utility Function

23 Another coroutine utility function we might see in C++23 or later is `when_any`, which enables a program to **wait for any of two or more tasks to complete**. `concurrencpp` provides a `when_any` function.

SE **One `when_any` use-case might be downloading several large files—one per task.** Though you might want all the results eventually, you'd like to start processing as soon as the first download is complete. You could then call `when_any` again for the remaining tasks that are still executing. `concurrencpp`'s `when_any` typically is used in a loop that keeps iterating until the last of several tasks completes.

`main` Program

Function `main` (lines 80–107) creates a `vector` containing 100 million random `int` values, calls `sortCoroutine` to sort the `vector`, waits for `sortCoroutine` to complete, then confirms that the `vector` is sorted.

```
80   int main() {
81      concurrencpp::runtime runtime; // set up concurrencpp runtime
82      auto executor{runtime.thread_pool_executor()}; // get the executor
83
84      // set up random number generation
85      std::random_device rd;
86      std::default_random_engine engine{rd()};
```

```
87      std::uniform_int_distribution ints;
88
89      std::cout << fmt::format(
90         "Thread {}: main creating vector of random ints\n", id());
91      std::vector<int> values(100'000'000);
92      std::ranges::generate(values, [&]() {return ints(engine);});
93
94      std::cout << fmt::format(
95         "Thread {}: main starting sortCoroutine\n", id());
96      auto result{sortCoroutine(executor, values)};
97
98      std::cout << fmt::format("\nThread {}: {}\n", id(),
99         "main resumed. Waiting for sortCoroutine to complete.");
100     result.get(); // wait for sortCoroutine to complete
101
102     std::cout << fmt::format(
103        "\nThread {}: main confirming that vector is sorted\n", id());
104     bool sorted{std::ranges::is_sorted(values)};
105     std::cout << fmt::format("Thread {}: values is{} sorted\n",
106        id(), sorted ? "" : " not");
107  }
```

The `main` function operates as follows:

- Lines 81–82 set up the **`concurrencpp::runtime`** object and get its **`concur-rencpp::thread_pool_executor`**.
- Lines 85–87 set up the random-number generation to populate a `vector<int>`.
- Lines 89–92 create the 100-million-element `vector<int>`, then fill it with random `int` values using the **`std::ranges::generate`** algorithm.
- Lines 94–95 display that the `main` thread is executing and about to call `sort-Coroutine`.
- Line 96 calls coroutine `sortCoroutine`, passing the `executor` and the `vector values`. This launches the asynchronous sorting tasks and returns an object of type **`concurrencpp::result<void>`**.

The `sortCoroutine` call executes in `main`'s thread to launch the **asynchronous tasks. When `sortCoroutine` `co_await`s those tasks, it will suspend its execution if their results are not yet available; otherwise, it will simply complete and return**.

- In this example, the asynchronous tasks will not be complete because it takes time for each task launched by `sortCoroutine` to sort 50 million elements. So, the coroutine will suspend its execution and return control to `main`. At this point, lines 98–99 will show that `main` is executing again by displaying a line of text. **This is where `main` could potentially continue doing other work.**
- Line 100 calls the **`result` object's `get` method to block `main` from continuing until `sortCoroutine`'s asynchronous tasks finish executing**.
- Once that happens, lines 102–103 display a message indicating that `main` is confirming `values` is sorted. Then, line 104 calls `std::ranges::is_sorted` with the `values` as an argument. This algorithm returns `true` if its argument is sorted; otherwise, it returns `false`. Finally, lines 105–106 display a message indicating whether the `vector` is sorted.

Sample Output

Let's discuss the sample output in detail. Below the output, we broke it into pieces for discussion purposes.

```
Thread 17276: main creating vector of random ints
```

```
Thread 17276: main starting sortCoroutine
Thread 17276: sortCoroutine started

Thread 17276: sortCoroutine starting first half sortTask
Thread 17276: sortCoroutine starting second half sortTask

Thread 17276: sortCoroutine co_awaiting sortTask completion

Thread 17276: main resumed. Waiting for sortCoroutine to complete.
Thread 17144: Sorting 50000000 elements
Thread 6252: Sorting 50000000 elements
Thread 6252: Finished sorting
Thread 17144: Finished sorting

Thread 6252: sortCoroutine merging results
Thread 6252: sortCoroutine done

Thread 17276: main confirming that vector is sorted
Thread 17276: values is sorted
```

First, `main` displays its thread ID and creates the `vector` of random integers:

```
Thread 17276: main creating vector of random ints
```

Next, `main` displays its thread ID and calls `sortCoroutine`:

```
Thread 17276: main starting sortCoroutine
```

Then, `sortCoroutine` displays its thread ID and indicates that it started:

```
Thread 17276: sortCoroutine started
```

Note that `sortCoroutine` executes in `main`'s thread, so it displays the same thread ID.

Next, `sortCoroutine` launches two tasks to sort the `vector`:

```
Thread 17276: sortCoroutine starting first half sortTask
Thread 17276: sortCoroutine starting second half sortTask
```

At this point, `sortCoroutine` **`co_await`s completion of the tasks**:

```
Thread 17276: sortCoroutine co_awaiting sortTask completion
```

If the `sortTask`s have not completed, **`sortCoroutine` suspends execution** and returns control to `main`. **We purposely created a large `vector` so the sorting would not complete immediately.** This enables us to show that `main` continues executing when the coroutine suspends execution:

```
Thread 17276: main resumed. Waiting for sortCoroutine to complete.
```

`main` displays the preceding line of text, then **calls the `sortCoroutine result` object's `get` function to block until `sortCoroutine` completes and returns control to `main`.**

In the meantime, the parallel `sortTask`s start running in other threads, as shown by their thread IDs:

```
Thread 17144: Sorting 50000000 elements
Thread 6252: Sorting 50000000 elements
```

We cannot predict the relative speeds of asynchronous concurrent tasks, so we do not know which `sortTask` will finish first. In this run, the `sortTask`s finished in the reverse of the order they started, as confirmed by their thread IDs:

```
Thread 6252: Finished sorting
Thread 17144: Finished sorting
```

Next, `sortCoroutine` merges the results of the two `sortTask`s, then indicates that it's done:

```
Thread 17144: sortCoroutine merging results
Thread 17144: sortCoroutine done
```

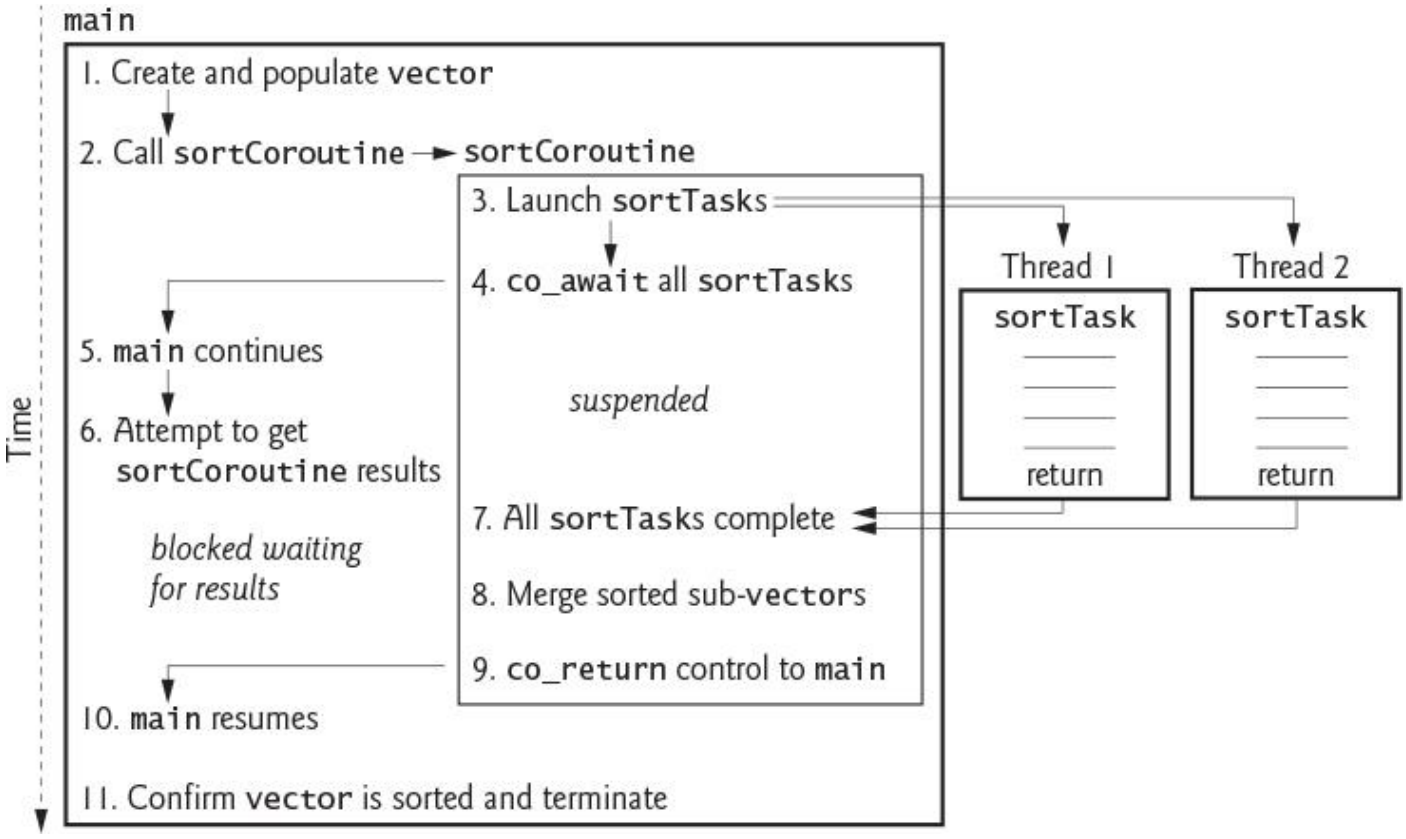
The executor resumed `sortCoroutine` on the thread of the last `sortTask` to complete.

At this point, `sortCoroutine` **`co_return`s** control to `main`, which continues executing, confirms that the `vector` is sorted, then terminates:

```
Thread 17276: main confirming that vector is sorted
Thread 17276: values is sorted
```

Flow of Control for a Coroutine with `co_await` and `co_return`

The following diagram shows the basic flow of control in this program—the numbers in this description correspond to the steps in the diagram:



1. `main` creates and populates the `vector` of random integers.
2. `main` calls `sortCoroutine`.
3. `sortCoroutine` launches two parallel tasks that each call `sortTask` for half the `vector`. Those tasks begin running in parallel.
4. `sortCoroutine co_await`s all of the tasks' results, suspending execution of the coroutine and returning control to `main`.
5. `main` continues executing.
6. `main` attempts to get `sortCoroutine`'s result. This blocks `main`'s from continuing until `sortCoroutine` completes execution and returns control to `main`.
7. Eventually, the `sortTask`s complete, at which point `sortCoroutine` resumes execution.
8. `sortCoroutine` merges the sorted sub-`vector`s.
9. `sortCoroutine co_return`s control to `main`, terminating `sortCoroutine`.
10. `main` resumes execution.
11. `main` confirms the `vector` is sorted and terminates.

18.7 Low-Level Coroutines Concepts

You've now seen that coroutine support libraries, like **`generator`** and **`concurrencpp`**, make it convenient to create coroutines. For programmers who want to try programming coroutines without using these "experimental" coroutine support libraries, this section overviews some of the low-level concepts. For further study, we've included key articles and videos with code examples in the footnotes.[32,33,34,35,36,37]

32. Marius Bancila, *Modern C++ Programming Cookbook*, Chapter 12, pp. 681–700. Packet, 2020.

33. Simon Toth, "C++20 Coroutines: Complete Guide," September 26, 2021. Accessed February 12, 2022. https://itnext.io/c-20-coroutines-complete-guide-7c3fc08db89d.

34. Rainer Grimm, "40 Years of Evolution from Functions to Coroutines," September 25, 2020. Accessed February 12, 2022. https://www.youtube.com/watch?v=jd6P9X8l2bY.

35. Jeff Thomas, “Exploring Coroutines,” April 7, 2021. Accessed February 12, 2022. https://blog.coffeetocode.com/2021/04/exploring-coroutines/.
36. Panicsoftware, “Your First Coroutine,” March 6, 2019. Accessed February 12, 2022. https://blog.panicsoftware.com/your-first-coroutine/.
37. Panicsoftware, “`co_await`ing Coroutines,” April 12, 2019. Accessed February 12, 2022. https://blog.panicsoftware.com/co_awaiting-coroutines/.

This section also will strengthen your understanding of the high-level, library-based approach we took in the preceding subsections. The details for topics discussed in this section can be found in C++ Standard[38] sections 7.6.2.3, 7.6.17, 8.7.4, 9.5.4 and 17.12, and at cppreference.com.[39, 40]

38. “C++ Standard,” Accessed February 12, 2022. https://timsong-cpp.github.io/cppwp/n4861/.
39. “Coroutines (C++20).” Accessed February 12, 2022. https://en.cppreference.com/w/cpp/language/coroutines.
40. “Standard Library Header `<coroutine>`.” Accessed February 12, 2022. https://en.cppreference.com/w/cpp/header/coroutine.

Coroutine Restrictions

20 According to the C++20 standard, the following kinds of functions cannot be coroutines:[41]

41. C++ Standard, Sections 6.9.3.1, 9.2.5, 9.2.8.5, 11.4.4, 11.4.6. Accessed February 12, 2022. https://timsong-cpp.github.io/cppwp/n4861/.

- `main`,
- constructors,
- destructors,
- `constexpr` and `consteval` functions,
- 20 functions declared with **placeholder types** (`auto` or **C++20 concepts**) and
- functions with variable numbers of arguments.

Promise Object

Every coroutine has a **promise object**[42] that’s created when the coroutine is first called. It’s used by the coroutine to

42. Lewis Baker, “C++ Coroutines: Understanding the Promise Type,” September 5, 2018. Accessed February 12, 2022. https://lewissbaker.github.io/2018/09/05/understanding-the-promise-type.

- return a result to its caller or
- return an exception to its caller.

The promise object is part of the coroutine state. It provides several member functions:

- `get_return_object`—When the coroutine begins executing, it calls this function on the **promise object** to get the coroutine’s **result object**, which will be returned to the coroutine’s caller when the coroutine first **suspends**. The caller uses the **result object** to access the coroutine’s result when it becomes available.
- `initial_suspend`—When the coroutine begins executing, it calls this function on the **promise object** to determine whether to immediately **suspend** (for **lazy coroutines**, like our `fibonacciGenerator` in Fig. 18.1) or **continue executing** (like our `sortCoroutine` coroutine in Fig. 18.3). The `<coroutine>` **header** provides types `suspend_always` and `suspend_never` to support these two possibilities.

- `unhandled_exception`—The coroutine calls this function on the **promise object** if an **unhandled exception** causes the coroutine to terminate.
- `return_void`—The coroutine calls this function on the **promise object** when execution reaches the coroutine's closing brace, a `co_return` statement with no expression or a `co_return` statement with an expression that evaluates to `void`. The **promise object** may define only `return_void` or `return_value`, but not both.
- `return_value`—The coroutine calls this function when a `co_return` statement contains a **non-`void` expression**. The **promise object** may define either `return_value` or `return_void`, but not both.
- `final_suspend`—After calling `return_void` or `return_value`, the coroutine calls this function to return control, and possibly a result, to the coroutine's caller.
- `yield_value`—When a `co_yield` statement executes, a **generator coroutine** calls this function to return a value to the coroutine's caller.

Coroutine State

The coroutine state (sometimes called the coroutine frame) maintains

- copies of the coroutine's **parameters**,
- its **promise object**,
- a representation of the **suspension point**, so the coroutine knows where to **resume execution**, and
- **local variables** in scope at the **suspension point**.

In our **generator** example (Fig. 18.1), the suspension point is where the `co_yield` statement **suspends the coroutine** and returns a value to the generator's caller. In our sorting example (Fig. 18.3), the **suspension point** is where we `co_await` the result of calling `concurrencpp::when_all`.

Coroutine Handle

Generator support libraries use a `coroutine_handle` (from **header** `<coroutine>`) to refer to and manipulate a **running** or **suspended** coroutine. Some capabilities of class template `coroutine_handle` include:

- creating a `coroutine_handle` from a **promise object**,
- checking whether the referenced coroutine has **completed execution**,
- **resuming a suspended coroutine**,
- **destroying the coroutine state** and
- getting the coroutine's **promise** object.

Awaitable Objects

The operand of `co_await` must be an awaitable object, such as a `concurrencpp::result` (Fig. 18.3). Such objects must provide several member functions:

- `await_ready`—When you `co_await` an **awaitable object**, this function is called first to **determine whether the result is already available**. If so, the coroutine continues executing.
- `await_suspend`—If the **result is not ready**, this function is called to **determine whether to suspend the coroutine's execution** and **return control to the caller**. If this function returns a `coroutine_handle` for another **suspended coroutine**, that **coroutine resumes execution**.
- `await_resume`—When the coroutine **resumes execution**, this function is called to **return the result** of the `co_await` **expression**.

18.8 C++23 Coroutines Enhancements

23 Section 18.5 demonstrated `concurrencpp` **executors** for launching tasks and managing threads. **Standard library executors** and other features to support coroutine development are expected in C++23. Various other third-party libraries have experimental executor implementations. For example, **Facebook's libunifex ("unified executors") library**[43] is a prototype implementation of **executors** and other asynchronous programming facilities currently being considered for standardization. For more information, see the C++ Standards Committee proposals:

- "`std::execution`,"[44]
- "A Unified Executors Proposal for C++"[45] and
- "Dependent Execution for a Unified Executors Proposal for C++."[46]

43. "libunifex." Accessed February 12, 2022. https://github.com/facebookexperimental/libunifex.
44. Michał Dominiak, Lewis Baker, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler and Bryce Adelstein Lelbach, "`std::execution`," October 4, 2021. Accessed February 12, 2022. https://wg21.link/p2300.
45. Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown, Daisy Hollman, Lee Howes, Kirk Shoop, Lewis Baker and Eric Niebler, "A Unified Executors Proposal for C++," September 15, 2020. Accessed February 12, 2022. http://wg21.link/p0443.
46. Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards and Gordon Brown, "Dependent Execution for a Unified Executors Proposal for C++," October 8, 2018. Accessed February 12, 2022. https://wg21.link/p1244.

18.9 Wrap-Up

This chapter presented a detailed higher-level treatment of coroutines—the last of C++20's "big four" features (ranges, concepts, modules and coroutines). The limited C++20 library support for coroutines is meant for developers creating higher-level coroutines support libraries for C++23 and later. We used this high-level, library-based approach to conveniently create coroutines, enabling sophisticated concurrent programming with a simple sequential-like coding style. We used keyword `co_yield` and the `generator` library to implement a lazy generator coroutine. We also introduced the `concurrencpp` library's executors, tasks and results, using them to execute tasks and define a coroutine with keywords `co_await` and `co_return`.

Thanks for reading *C++20 for Programmers*. We hope that you enjoyed the book and that you found it entertaining and informative. Most of all we hope you feel empowered to apply the technologies you've learned to the challenges you'll face in your career.

19. Stream I/O & C++20 Text Formatting

Objectives

In this chapter, you'll:

- Use C++ object-oriented stream input/output.
- Input and output individual characters.
- Use unformatted I/O for high performance.
- Use stream manipulators to display integers in octal and hexadecimal formats.
- Specify precision for input and output.
- Display floating-point values in scientific and fixed-point notation.
- Set and restore the format state.
- Control alignment and padding.
- Determine the success or failure of input/output operations.
- Tie output streams to input streams.
- See many of C++20's concise and convenient text-formatting capabilities, including presentation types to specify data types to format, positional arguments, field widths, alignment, numeric formatting and using placeholders to specify field widths and precisions.

Outline

19.1 Introduction
19.2 Streams
19.2.1 Classic Streams vs. Standard Streams
19.2.2 `iostream` Library Headers
19.2.3 Stream Input/Output Classes and Objects
19.3 Stream Output
19.3.1 Output of `char*` Variables
19.3.2 Character Output Using Member Function `put`
19.4 Stream Input
19.4.1 `get` and `getline` Member Functions
19.4.2 `istream` Member Functions `peek`, `putback` and `ignore`
19.5 Unformatted I/O Using `read`, `write` and `gcount`
19.6 Stream Manipulators
19.6.1 Integral Stream Base (`dec`, `oct`, `hex` and `setbase`)
19.6.2 Floating-Point Precision (`setprecision`, `precision`)
19.6.3 Field Width (`width`, `setw`)
19.6.4 User-Defined Output Stream Manipulators
19.6.5 Trailing Zeros and Decimal Points (`showpoint`)
19.6.6 Alignment (`left`, `right` and `internal`)
19.6.7 Padding (`fill`, `setfill`)
19.6.8 Integral Stream Base (`dec`, `oct`, `hex`, `showbase`)
19.6.9 Floating-Point Numbers; Scientific and Fixed Notation (`scientific`, `fixed`)
19.6.10 Uppercase/Lowercase Control (`uppercase`)
19.6.11 Specifying Boolean Format (`boolalpha`)
19.6.12 Setting and Resetting the Format State via Member Function `flags`
19.7 Stream Error States

19.8 Tying an Output Stream to an Input Stream
19.9 C++20 Text Formatting
19.9.1 C++20 `std::format` Presentation Types
19.9.2 C++20 `std::format` Field Widths and Alignment
19.9.3 C++20 `std::format` Numeric Formatting
19.9.4 C++20 `std::format` Field Width and Precision Placeholders
19.10 Wrap-Up

19.1 Introduction

20 This chapter discusses input/output formatting capabilities. First, we present the I/O streams formatting capabilities that you're likely to see in C++ legacy code. Then, we discuss features from C++20's new text-formatting capabilities. In Section 19.9, you'll see that C++20 text formatting is more concise and convenient than the I/O streams formatting capabilities presented in Section 19.6.

C++ uses **type-safe I/O**. Each I/O operation is executed in a manner sensitive to the data type. If an I/O function has been defined to handle a particular data type, then that function is called to handle that data type. If there is no match between the type of the actual data and a function for handling data of that type, the compiler generates an error. Thus, improper data cannot "sneak" through the system. As you saw in Chapter 11, you can specify how to perform I/O for objects of user-defined types by overloading the stream insertion operator (`<<`) and the stream extraction operator (`>>`).

19.2 Streams

C++ I/O occurs in **streams**, which are sequences of bytes. In input operations, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection) to main memory. In output operations, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection).

Perf An application associates meaning with bytes. The bytes could represent characters, raw data, graphics images, audio, video or other information an application requires. The system I/O mechanisms transfer bytes from devices to memory and vice versa. The time these transfers take typically is far greater than the time the processor requires to manipulate data in memory. I/O operations require careful planning and tuning to ensure optimal performance.

C++ provides both "low-level" and "high-level" I/O capabilities. Low-level **unformatted I/O** capabilities specify that some number of bytes should be transferred device-to-memory or memory-to-device. In such transfers, the individual byte is the item of interest. Such low-level capabilities provide high-speed, high-volume transfers but are not particularly convenient. Higher-level **formatted I/O** groups bytes into meaningful units, such as integers, floating-point numbers, characters, strings and custom types.

19.2.1 Classic Streams vs. Standard Streams

C++'s **classic stream libraries** originally supported only `char`-based I/O. Because a `char` occupies one byte, it can represent only a limited set of characters, such as those in the ASCII character set. Many languages use alphabets that contain more characters than a single-byte `char` can represent. Such

characters are typically available in the extensive international **Unicode® character set** (https://unicode.org), which can represent most of the world's languages, mathematical symbols, emoji characters and more. C++ supports Unicode via the types

- `wchar_t`,
- 11 `char16_t` and `char32_t` (both from C++11), and
- 20 `char8_t` (C++20).

The **standard stream library classes** are class templates that can be instantiated for these various character types. We use the predefined stream-library instantiations for type `char` in this book. Unicode-based applications would use appropriate class-template instantiations based on the preceding types. C++ also supports Unicode string literals—for more information, see

```
https://en.cppreference.com/w/cpp/language/string_literal
```

19.2.2 `iostream` Library Headers

20 The C++ stream libraries provide hundreds of I/O capabilities. Most of our C++ programs include the `<iostream>` header, which declares basic services required for all stream-I/O operations.
The `<iostream>` header defines the **`cin`**, **`cout`**, **`cerr`** and **`clog`** objects, which correspond to the **standard input stream**, the **standard output stream**, the **unbuffered standard error stream** and the **buffered standard error stream**, respectively. The next section discusses `cerr`, `clog` and buffering.
The `<iostream>` and `<iomanip>` headers define **stream manipulators** for formatted I/O. We'll demonstrate many of these in this chapter. We'll also show that the newer C++20 text formatting with the `format` function greatly simplifies formatting output.

19.2.3 Stream Input/Output Classes and Objects

This chapter focuses on the `iostream` class templates:

- **`basic_istream`** for stream input operations and
- **`basic_ostream`** for stream output operations.

Though we do not use it in this chapter, **`basic_iostream`** provides stream input and stream output operations.

For each of the class templates `basic_istream`, `basic_ostream` and `basic_iostream`, the `iostream` library defines a type alias for `char`-based I/O:

- **`istream`** is a `basic_istream<char>` for `char` input—this is `cin`'s type.
- **`ostream`** is a `basic_ostream<char>` for `char` output—this is the type of `cout`, `cerr` and `clog`.
- **`iostream`** is a `basic_iostream<char>` for `char` input and output.

We used the aliases `istream` and `ostream` in Chapter 11 when we overloaded the stream extraction and stream insertion operators. The library also defines versions of these for `wchar_t`-based I/O—named `wistream`, `wostream` and `wiostream`, respectively. We cover only the `char`-based streams here.

Standard Stream Objects `cin`, `cout`, `cerr` and `clog`

Predefined object `cin` is an `istream` that's connected to the **standard input device**—usually the keyboard. In a stream extraction operation (`>>`) like

```
int grade{0};
```

```
std::cin >> grade; // data "flows" in the direction of the arrows
```

the compiler selects the appropriate overloaded stream extraction operator, based on the type of the variable `grade`—the one for `int` in this case. The `>>` operator is overloaded to input fundamental-type values, strings and pointer values.

The predefined object `cout` is an `ostream` that's connected to the **standard output device**. Standard output typically appears in

- a Command Prompt or PowerShell window in Microsoft Windows,
- a Terminal in macOS or Linux, or
- a shell window in Linux.

The stream insertion operator (`<<`) outputs its right operand to the standard output device:

```
std::cout << grade; // data "flows" in the direction of the arrows
```

The compiler selects the appropriate stream insertion operator for `grade`'s type—`<<` is overloaded to output data items of fundamental types, strings and pointer values.

The predefined object `cerr` is an `ostream` that's connected to the **standard error device**—typically the same device as the standard output device. Outputs to object `cerr` are **unbuffered**, meaning that each stream insertion to `cerr` performs its output immediately—this is appropriate for notifying a user promptly about errors.

Perf The predefined object `clog` is an `ostream` that's connected to the **standard error device**. Outputs to `clog` are **buffered**. Each output might be held in a buffer (that is, an area in memory) until the buffer is filled or until the buffer is flushed. Buffering is an I/O performance-enhancement technique.

19.3 Stream Output

`ostream` provides both formatted and unformatted output capabilities, including

- outputting standard data types with the stream insertion operator (`<<`);
- outputting characters via the `put` member function;
- **unformatted output** via the `write` member function;
- outputting integers in decimal, octal and hexadecimal formats;
- outputting floating-point values with various precisions, with **forced decimal points**, in **scientific notation** (e.g., `1.234567e-03`) and in **fixed notation** (e.g., `0.00123457`);
- outputting data **aligned** in fields of designated widths;
- outputting data in fields **padded** with specified characters; and
- outputting uppercase letters in scientific notation and **hexadecimal (base-16) notation**.

We'll demonstrate all of these capabilities in this chapter.

19.3.1 Output of `char*` Variables

Generally, you should avoid using pointers in favor of the modern C++ techniques we've discussed. In programs that require pointers, occasionally, you might want to print the addresses they contain (e.g., for debugging). The `<<` operator outputs a `char*` as a **null-terminated C-style string**. To output the **address**, cast the `char*` to a `void*` (Fig. 19.1, line 12). Operator <<'s `void*` version displays the pointer in an implementation-dependent manner—often as a hexadecimal number.[1] Figure 19.1 prints a `char*` variable as a null-terminated C-style string and an address. The output will vary by compiler

and operating system. We say more about controlling the bases of numbers in Section 19.6.1 and Section 19.6.8.

---

1. To learn more about hexadecimal numbers, see online Appendix C, Number Systems.

```
// fig19_01.cpp
// Printing the address stored in a char* variable.
#include <iostream>

int main() {
   const char* const word{"again"};

   // display the value of char* variable word, then display
   // the value of word after a static_cast to void*
   std::cout << "Value of word is: " << word
      << "\nValue of static_cast<const void*>(word) is: "
      << static_cast<const void*>(word) << '\n';
}
```

```
Value of word is: again
Value of static_cast<const void*>(word) is: 00007FF611416410
```

Fig. 19.1 Printing the address stored in a `char*` variable.

19.3.2 Character Output Using Member Function `put`

The `basic_ostream` member function `put` outputs one character at a time. For example, the statement

```
std::cout.put('A');
```

displays a single character `A`. Calls to `put` can be chained, as in

```
std::cout.put('A').put('\n');
```

which outputs the letter `A` followed by a newline character. As with `<<`, the preceding statement executes in this manner because the dot operator (`.`) groups left-to-right, and the `put` member function returns a reference to the `ostream` object (`cout`) that received the `put` call. You also can call `put` with a numeric expression representing a character value. For example, the following statement outputs uppercase `A`:

```
std::cout.put(65);
```

19.4 Stream Input

`istream` provides formatted and unformatted input capabilities. The stream extraction operator (`>>`) normally skips white-space characters (such as blanks, tabs and newlines) in the input stream. Later, we'll see how to change this behavior.

Using the Result of a Stream Extraction as a Condition

11 After each input, the stream extraction operator returns a reference to the stream object that received the extraction message (e.g., `cin` in the expression `cin >> grade`). If that reference is used as a condition (e.g., in a `while` statement's loop-continuation condition), the stream's overloaded `bool` cast operator function (C++11) is implicitly invoked to convert the reference into `true` or `false` value, based on the success or failure, respectively, of the last input operation. When an attempt is made to read past the end of a stream, the stream's overloaded `bool` cast operator returns `false` to indicate end-of-file. We used this capability in line 24 of Fig. 4.6.

19.4.1 `get` and `getline` Member Functions

The **`get`** member function with no arguments inputs and returns one character from the designated stream—including white-space characters and other nongraphic characters, such as the key sequence that represents end-of-file. This version of `get` returns `EOF` when end-of-file is encountered on the stream. `EOF` normally has the value `-1` and is defined in a header that's included in your code via stream library headers like `<iostream>`.

Using Member Functions `eof`, `get` and `put`

Figure 19.2 demonstrates member functions `eof` and `get` on input stream `cin` and member function `put` on output stream `cout`. This program uses `get` to read characters into the `int` variable `character`, so we can test for `EOF`. Function `get` returns an `int` because **`char` can represent only nonnegative values on many platforms**, and **`EOF` is typically defined as `-1`**. Line 10 prints the value of `cin.eof()`—initially `false`—before any inputs to show that end-of-file has not yet occurred on `cin`. You enter a line of text and press *Enter* followed by the end-of-file indicator:

- *<Ctrl> z* on Microsoft Windows systems or
- *<Ctrl> d* on Linux and Mac systems.

Line 14 reads each character, which line 15 outputs to `cout` using member function `put`. When end-of-file is encountered, the `while` statement ends, and lines 19–20 display the integer value of the last character read (`-1` for end-of-file) and the current value of `cin.eof()`, which now returns `true`, to show that end-of-file has been set on `cin`. Function `eof` returns `true` only after the program attempts to read past the last character in the stream.

```
1  // fig19_02.cpp
2  // get, put and eof member functions.
3  #include <format>
4  #include <iostream>
5
6  int main() {
7     int character{0}; // use int, because char cannot represent EOF
8
9     // prompt user to enter line of text
10    std::cout << std::format("Before input, cin.eof(): {}", std::cin.eof())
11       << "\nEnter a sentence followed by Enter and end-of-file:\n";
12
13    // use get to read each character; use put to display it
14    while ((character = std::cin.get()) != EOF) {
15       std::cout.put(character);
16    }
17
```

```
18      // display end-of-file character
19      std::cout << std::format("\nEOF on this system is: {}\n", character)
20         << std::format("After EOF input, cin.eof(): {}\n", std::cin.eof());
21   }
```

```
Before input, cin.eof(): false
Enter a sentence followed by Enter and end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z

EOF on this system is: -1
After EOF input, cin.eof(): true
```

Fig. 19.2 `get`, `put` and `eof` member functions.

Other `get` Versions

The `get` member function with a character-reference argument inputs the next character from the input stream and stores it in the character argument. This version of `get` returns a reference to the `istream` object on which the function is invoked.

Err A third version of `get` takes three arguments—a built-in array of `char`s, a size limit and a delimiter (with default value `'\n'`). This version can read multiple characters from the input stream. It either reads one fewer than the specified maximum number of characters and terminates or terminates as soon as the delimiter is read. A null character is inserted to terminate the input string in the character array argument. The delimiter is not placed in the character array. Rather, it remains in the input stream and will be the next character read if the program performs more input. Thus, the result of a second consecutive `get` is an empty line (possibly a logic error) unless the delimiter character is removed from the input stream—which you can do simply by calling `cin.ignore()`.

Comparing `cin` and `cin.get`

Figure 19.3 compares input using the stream extraction operator with `cin` (line 14), which reads characters until a white-space character is encountered, and input using the three-argument version of `cin.get` with its third argument defaulted to the `'\n'` character.

```
1   // fig19_03.cpp
2   // Contrasting input of a string via cin and cin.get.
3   #include <format>
4   #include <iostream>
5
6   int main() {
7      // create two char arrays, each with 80 elements
8      constexpr int size{80};
9      char buffer1[size]{};
10     char buffer2[size]{};
11
12     // use cin to input characters into buffer1
13     std::cout << "Enter a sentence:\n";
14     std::cin >> buffer1;
15
16     // display buffer1 contents
17     std::cout << std::format("\nThe cin input was:\n{}\n\n", buffer1);
```

```
18
19      // use cin.get to input characters into buffer2
20      std::cin.get(buffer2, size);
21
22      // display buffer2 contents
23      std::cout << std::format("The cin.get input was:\n{}\n", buffer2);
24   }
```

```
Enter a sentence:
Contrasting string input with cin and cin.get

The cin input was:
Contrasting

The cin.get input was:
  string input with cin and cin.get
```

Fig. 19.3 Contrasting input of a string via `cin` and `cin.get`.

Err 20 Before C++20, line 14 could write past the end of `buffer1`—a potentially fatal logic error. In C++20, the `char` array overload of `operator>>` is a **function template** that the compiler instantiates using its `char` array argument's size. In line 14, the compiler knows that `buffer1` contains 80 characters (as defined in line 9), so it instantiates an `operator>>` function that limits the number of characters input to a maximum of 79, saving one array element for the C-style string's terminating `'\0'` character.

Using Member Function `getline`

Member function **`getline`** operates similarly to the third version of the `get` member function and inserts a null character after the line in the built-in array of `char`s. The `getline` function removes the delimiter from the stream (i.e., reads the character and discards it) but **does not store it in the character array**. The program of Fig. 19.4 uses `getline` to input a line of text (line 12). Again, you should avoid using built-in arrays and pointers. So, C++ also provides `std::getline` in the `<string>` header.[2] This version of `getline` reads data and places it into a `string` object.

2. "`std::getline`." Accessed March 29, 2022. https://en.cppreference.com/w/cpp/string/basic_string/getline.

```
1   // fig19_04.cpp
2   // Inputting characters using cin member function getline.
3   #include <format>
4   #include <iostream>
5
6   int main() {
7      const int size{80};
8      char buffer[size]{}; // create array of 80 characters
9
10     // input characters in buffer via cin function getline
11     std::cout << "Enter a sentence:\n";
12     std::cin.getline(buffer, size);
13
14     // display buffer contents
15     std::cout << std::format("\nYou entered:\n{}\n", buffer);
16  }
```

```
Enter a sentence:
```

```
Using the getline member function

You entered:
Using the getline member function
```

Fig. 19.4 Inputting characters using `cin` member function `getline`.

19.4.2 `istream` Member Functions `peek`, `putback` and `ignore`

`istream` member function `ignore` reads and discards characters. It receives as arguments:

- a designated number of characters—the default argument value is 1—and
- a delimiter at which to stop ignoring characters—the default delimiter is `EOF`.

The function discards the specified number of characters, or fewer characters if the delimiter is encountered in the input stream.

The **`putback`** member function places the previous character obtained by a `get` from an input stream back into that stream. This is helpful in applications that scan an input stream looking for a field beginning with a specific character. When that character is input, the application returns the character to the stream for the next input operation.

The **`peek`** member function returns the next character from an input stream but does not remove the character from the stream.

19.5 Unformatted I/O Using `read`, `write` and `gcount`

Unformatted input/output is performed using `istream`'s **`read`** and `ostream`'s **`write`** member functions, respectively:

- `read` inputs bytes to a built-in array of `char`s in memory.
- `write` outputs bytes from a built-in array of `char`s.

These bytes are input or output simply as "raw" bytes—they are not formatted in any way. For example, the following call outputs the first 10 bytes of `buffer`, including null characters, if any, that would cause output with `cout` and `<<` to terminate:

```
char buffer[]{"HAPPY BIRTHDAY"};
std::cout.write(buffer, 10);
```

Similarly, the following call displays the first 10 characters of the alphabet:

```
std::cout.write("ABCDEFGHIJKLMNOPQRSTUVWXYZ", 10);
```

The `read` member function inputs a designated number of characters into a built-in array of `char`s. If fewer than the designated number of characters are read, `failbit` is set. Section 19.7 shows how to determine whether `failbit` has been set. Member function **`gcount`** reports the number of characters read by the last input operation.

Figure 19.5 demonstrates `istream` member functions `read` and `gcount`, and `ostream` member function `write`. The program inputs 20 characters (from a longer input sequence) into the array `buffer` with `read` (line 10), determines the number of characters input with `gcount` (line 14) and outputs the characters in `buffer` with `write` (line 14).

```
1   // fig19_05.cpp
2   // Unformatted I/O using read, gcount and write.
3   #include <iostream>
4
5   int main() {
6      char buffer[80]{}; // create array of 80 characters
7
8      // use function read to input characters into buffer
9      std::cout << "Enter a sentence:\n";
10     std::cin.read(buffer, 20);
11
12     // use functions write and gcount to display buffer characters
13     std::cout << "\nThe sentence entered was:\n";
14     std::cout.write(buffer, std::cin.gcount());
15     std::cout << '\n';
16  }
```

```
Enter a sentence:
Using the read, write, and gcount member functions

The sentence entered was:
Using the read, writ
```

Fig. 19.5 Unformatted I/O using `read`, `gcount` and `write`.

19.6 Stream Manipulators

C++ provides various **stream manipulators** to specify formatting in streams. The stream manipulators provide capabilities such as

- setting the base for integer values
- setting field widths
- setting precision
- setting and unsetting format state
- setting the fill character in fields
- flushing streams
- inserting a newline into the output stream (and flushing the stream)
- inserting a null character into the output stream
- skipping white space in the input stream

The following table lists various stream manipulators that control a given stream's format state. We show examples of many of these stream manipulators in the next several sections.

Manipulator	Description
`skipws`	**Skips white-space characters** on an input stream. You can reset this setting with stream manipulator `noskipws`.
`left`	**Left aligns** output in a field. Padding characters appear to the right if necessary.

Manipulator	Description
`right`	**Right aligns** output in a field. Padding characters appear to the left if necessary.
`internal`	In a field, this **left aligns a number's sign** and **right aligns a number's value**. Padding characters appear between the sign and the number if necessary.
`boolalpha`	Displays `bool` values as the word `true` or `false`. Similarly, `noboolalpha` sets the stream back to displaying bool values as 1 (true) and 0 (false).
`dec`	Treats integers as **decimal** (base 10) values.
`oct`	Treats integers as **octal** (base 8) values.
`hex`	Treats integers as **hexadecimal** (base 16) values.
`showbase`	Outputs a number's **base** before the number—`0` for octals and `0x` or `0X` for hexadecimals. You can reset this with stream manipulator `noshowbase`.
`showpoint`	Forces floating-point numbers do display a decimal point. Normally, this is used with `fixed` to guarantee a certain number of digits to the right of the decimal point. You can reset this setting with stream manipulator `noshowpoint`.
`uppercase`	Displays hexadecimal integers with uppercase letters (i.e., `X` and `A` through `F`) and displays floating-point values in scientific notation with an uppercase `E`. You can reset this setting with stream manipulator `nouppercase`.
`showpos`	Precedes positive numbers by a plus sign (**+**). You can reset this with `noshowpos`.
`scientific`	Outputs floating-point values in **scientific notation**.
`fixed`	Outputs floating-point values in **fixed-point notation** with a specific number of digits to the right of the decimal point.

19.6.1 Integral Stream Base: `dec`, `oct`, `hex` and `setbase`

Integers typically are processed as decimal (base 10) values. To change this, you can insert the **`hex`** stream manipulator to set the base to hexadecimal (base 16) or insert the **`oct`** manipulator to set the base to octal (base 8). Insert the **`dec`** manipulator to reset the stream base to decimal. These **stream manipulators** are all **sticky**—that is, the settings remain in effect until you change them.

You also can set a stream's integer base via the **`setbase` parameterized stream manipulator** (header `<iomanip>`). A parameterized stream manipulator takes an argument—in this case, the value `10`, `8`, or `16` to set the base to decimal, octal or hexadecimal.[3] The stream base value remains the same until changed explicitly, so `setbase` settings are sticky. Figure 19.6 demonstrates stream manipulators `hex` (line 14), `dec` (line 17), `oct` (line 18) and `setbase` (line 21).

3. Appendix C, Number Systems, discusses the decimal, octal and hexadecimal number systems.

```
1  // fig19_06.cpp
2  // Using stream manipulators dec, oct, hex and setbase.
3  #include <iomanip>
4  #include <iostream>
5
6  int main() {
7     int number{0};
8
9     std::cout << "Enter a decimal number: ";
10    std::cin >> number; // input number
11
12    // use hex stream manipulator to show hexadecimal number
13    std::cout << number << " in hexadecimal is: "
14       << std::hex << number << "\n";
15
16    // use oct stream manipulator to show octal number
17    std::cout << std::dec << number << " in octal is: "
18       << std::oct << number << "\n";
19
20    // use setbase stream manipulator to show decimal number
21    std::cout << std::setbase(10) << number << " in decimal is: "
22       << number << "\n";
23 }
```

```
Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20
```

Fig. 19.6 Using stream manipulators `dec`, `oct`, `hex` and `setbase`.

19.6.2 Floating-Point Precision (`setprecision`, `precision`)

You can control the **precision** of floating-point numbers—that is, the number of digits to the right of the decimal point—with the **`setprecision`** stream manipulator or the `ost-ream` member function **`precision`**. Both are sticky—a call to either sets the precision for all subsequent output

operations until the next precision-setting call. Calling member function `precision` with no argument returns the current precision setting. You can use this to save the current precision setting so you can restore it later. Figure 19.7 uses both member function `precision` (line 16) and the `setprecision` manipulator (line 24) to print a table that shows the square root of `2`, with precision varying from `0` to `9`. Stream manipulator `fixed` (line 12) forces a floating-point number to display in fixed-point notation with a specific number of digits to the right of the decimal point, as specified by member function `precision` or stream manipulator `setprecision`.

```
1  // fig19_07.cpp
2  // Controlling precision of floating-point values.
3  #include <iomanip>
4  #include <iostream>
5  #include <cmath>
6
7  int main() {
8     double root2{std::sqrt(2.0)}; // calculate square root of 2
9
10    std::cout << "Square root of 2 with precisions 0-9.\n"
11       << "Precision set by ostream member function precision:\n";
12    std::cout << std::fixed; // use fixed-point notation
13
14    // display square root using ostream function precision
15    for (int places{0}; places <= 9; ++places) {
16       std::cout.precision(places);
17       std::cout << root2 << "\n";
18    }
19
20    std::cout << "\nPrecision set by stream manipulator setprecision:\n";
21
22    // set precision for each digit, then display square root
23    for (int places{0}; places <= 9; ++places) {
24       std::cout << std::setprecision(places) << root2 << "\n";
25    }
26 }
```

```
Square root of 2 with precisions 0-9.
Precision set by ostream member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
```

```
1.41421
1.414214
1.4142136
1.41421356
1.414213562
```

Fig. 19.7 Controlling precision of floating-point values.

19.6.3 Field Width (`width`, `setw`)

The **`width`** member function (of classes `istream` and `ostream`) sets the **field width**—that is, the number of character positions in which a value should be output or the maximum number of characters that should be input. The function also returns the previous field width so you can save it and restore the value later. If the values output are narrower than the field width, **fill characters** are inserted as **padding**. When a field is not sufficiently wide to handle outputs, the outputs print as wide as necessary, which can yield confusing outputs. **The width setting is not sticky—it applies only for the next insertion or extraction**. Afterward, the width is set implicitly to `0`, so subsequent inputs or outputs will be performed with default settings. Calling `width` with no argument returns the current setting.

Figure 19.8 demonstrates the `width` member function for both input (lines 10 and 16) and output (line 14). For input into a `char` array, a maximum of **one fewer characters than the width are read**, saving one element for the null character to be placed at the end of the C-style string. Remember that stream extraction terminates when nonleading white space is encountered. When prompted for input in Fig. 19.8, enter a line of text and press *Enter* followed by end-of-file (*<Ctrl> z* on Microsoft Windows systems and *<Ctrl> d* on Linux and OS X systems). The **`setw` parameterized stream manipulator** also may be used to set the field width by inserting a call to it in a `cin` or `cout` statement.

```
1  // fig19_08.cpp
2  // width member function of classes istream and ostream.
3  #include <iostream>
4
5  int main() {
6     int widthValue{4};
7     char sentence[10]{};
8
9     std::cout << "Enter a sentence:\n";
10    std::cin.width(5); // input up to 4 characters from sentence
11
12    // set field width, then display characters based on that width
13    while (std::cin >> sentence) {
14       std::cout.width(widthValue++);
15       std::cout << sentence << "\n";
16       std::cin.width(5); // input up to 4 more characters from sentence
17    }
18 }
```

```
Enter a sentence:
This is a test of the width member function
This
   is
     a
   test
      of
```

```
      the
      widt
          h
        memb
           er
          func
           tion
^Z
```

Fig. 19.8 `width` member function of class classes `istream` and `ostream`.

19.6.4 User-Defined Output Stream Manipulators

You can create your own stream manipulators. Figure 19.9 shows how to create and use custom nonparameterized output-stream manipulators `bell` (lines 7–9) and `tab` (lines 12–14). These are defined as functions with `ostream&` as the return type and parameter type. When lines 19 and 21 insert `tab` and `bell` in the output stream, their corresponding functions are called, which in turn output the `\a` (alert) and `\t` (tab) escape sequences, respectively. The `bell` manipulator does not display any text. Rather, it plays your system's alert sound.

```
1   // fig19_09.cpp
2   // Creating and testing user-defined, nonparameterized
3   // stream manipulators.
4   #include <iostream>
5
6   // bell manipulator (using escape sequence \a)
7   std::ostream& bell(std::ostream& output) {
8      return output << '\a'; // issue system beep
9   }
10
11  // tab manipulator (using escape sequence \t)
12  std::ostream& tab(std::ostream& output) {
13     return output << '\t'; // issue tab
14  }
15
16  int main() {
17     // use tab and bell manipulators
18     std::cout << "Testing the tab manipulator:\n"
19        << 'a' << tab << 'b' << tab << 'c' << '\n';
20
21     std::cout << "Testing the bell manipulator\n" << bell;
22  }
```

```
Testing the tab manipulator:
a       b       c
Testing the bell manipulator
```

Fig. 19.9 Creating and testing user-defined, nonparameterized stream manipulators.

19.6.5 Trailing Zeros and Decimal Points (`showpoint`)

Stream manipulator **`showpoint`** (Fig. 19.10) is a sticky setting that forces a floating-point number to display a decimal point and trailing zeros. For example, the floating-point value `79.0` prints as `79` without using `showpoint` and prints as `79.00000` using `showpoint`. The number of trailing zeros is determined by the current **precision**. To reset the `showpoint` setting, output the stream

manipulator `noshowpoint`. The **default precision** of floating-point numbers is 6, which you can see in the output produced by lines 13–17. When neither the `fixed` nor the `scientific` stream manipulator is used, the precision represents the number of significant digits to display (i.e., the total number of digits to display), not the number of digits to display after the decimal point.

```
1  // fig19_10.cpp
2  // Displaying trailing zeros and decimal points in floating-point values.
3  #include <iostream>
4
5  int main() {
6     // display double values with default stream format
7     std::cout << "Before using showpoint"
8        << "\n9.9900 prints as: " << 9.9900
9        << "\n9.9000 prints as: " << 9.9000
10       << "\n9.0000 prints as: " << 9.0000;
11
12    // display double value after showpoint
13    std::cout << std::showpoint
14       << "\n\nAfter using showpoint"
15       << "\n9.9900 prints as: " << 9.9900
16       << "\n9.9000 prints as: " << 9.9000
17       << "\n9.0000 prints as: " << 9.0000 << '\n';
18 }
```

```
Before using showpoint
9.9900 prints as: 9.99
9.9000 prints as: 9.9
9.0000 prints as: 9

After using showpoint
9.9900 prints as: 9.99000
9.9000 prints as: 9.90000
9.0000 prints as: 9.00000
```

Fig. 19.10 Controlling the printing of trailing zeros and decimal points in floating-point values.

19.6.6 Alignment (`left`, `right` and `internal`)

Stream manipulators `left` and `right` enable fields to be **left-aligned** with **padding** characters to the right or **right-aligned** with **padding** characters to the left, respectively. The padding character is specified by the `fill` member function or the `setfill` parameterized stream manipulator (which we discuss in Section 19.6.7). Figure 19.11 uses the `setw`, `left` and `right` manipulators to left align and right align integer data in a field—we wrap each field in quotes so you can see the leading and trailing space in the field.

```
1  // fig19_11.cpp
2  // Left and right alignment with stream manipulators left and right.
3  #include <iomanip>
4  #include <iostream>
5
6  int main() {
7     int x{12345};
8
```

```
9       // display x right aligned (default)
10      std::cout << "Default is right aligned:\n\""
11         << std::setw(10) << x << "\"";
12
13      // use left manipulator to display x left aligned
14      std::cout << "\n\nUse left to left align x:\n\""
15         << std::left << std::setw(10) << x << "\"";
16
17      // use right manipulator to display x right aligned
18      std::cout << "\n\nUse right to right align x:\n\""
19         << std::right << std::setw(10) << x << "\"\n";
20   }
```

```
Default is right aligned:
"     12345"

Use left to left align x:
"12345     "

Use right to right align x:
"     12345"
```

Fig. 19.11 Left and right alignment with stream manipulators `left` and `right`.

Stream manipulator `internal` (Fig. 19.12; line 8) indicates that a number's **sign** should be **left-aligned** within a field, the number's magnitude should be **right-aligned** and intervening spaces should be padded with the **fill character**. When using stream manipulator `showbase`, the **base** is left aligned. The `showpos` manipulator (line 8) forces the plus sign to print. To reset the `showpos` setting, output the stream manipulator `noshowpos`.

```
1   // fig19_12.cpp
2   // Printing an integer with internal spacing and plus sign.
3   #include <iomanip>
4   #include <iostream>
5
6   int main() {
7      // display value with internal spacing and plus sign
8      std::cout << std::internal << std::showpos
9         << std::setw(10) << 123 << "\n";
10  }
```

```
+       123
```

Fig. 19.12 Printing an integer with internal spacing and plus sign.

19.6.7 Padding (`fill`, `setfill`)

The `fill` member function specifies the **fill character** to use with aligned fields and returns the previous fill character. Spaces are used for padding by default. The `setfill` manipulator also sets the **padding character**. Figure 19.13 demonstrates `fill` (line 29) and `setfill` (lines 33 and 38) to set the fill character.

```
1   // fig19_13.cpp
2   // Using member function fill and stream manipulator setfill to change
3   // the padding character for fields larger than the printed value.
4   #include <iomanip>
```

```
#include <iostream>

int main() {
   int x{10000};

   // display x
   std::cout << x << " printed as int right and left aligned\n"
      << "and as hex with internal alignment.\n"
      << "Using the default pad character (space):\n";

   // display x
   std::cout << std::setw(10) << x << "\n";

   // display x with left alignment
   std::cout << std::left << std::setw(10) << x << "\n";

   // display x with base as hex with internal alignment
   std::cout << std::showbase << std::internal << std::setw(10)
      << std::hex << x << "\n\n";

   std::cout << "Using various padding characters:\n";

   // display x using padded characters (right alignment)
   std::cout << std::right;
   std::cout.fill('*');
   std::cout << std::setw(10) << std::dec << x << "\n";

   // display x using padded characters (left alignment)
   std::cout << std::left << std::setw(10) << std::setfill('%')
      << x << "\n";

   // display x using padded characters (internal alignment)
   std::cout << std::internal << std::setw(10)
      << std::setfill('^') << std::hex << x << "\n";
}
```

```
10000 printed as int right and left aligned
and as hex with internal alignment.
Using the default pad character (space):
     10000
10000
0x    2710

Using various padding characters:
*****10000
10000%%%%%
0x^^^^2710
```

Fig. 19.13 Using member function `fill` and stream manipulator `setfill` to change the padding character for fields larger than the printed values.

19.6.8 Integral Stream Base (`dec`, `oct`, `hex`, `showbase`)

C++ provides stream manipulators **`dec`**, **`oct`** and **`hex`** to specify that integers should display as decimal, hexadecimal and octal values, respectively. Integers display in decimal (base 10) by default. With stream extraction, integers prefixed with `0` are treated as octal values, integers prefixed

with `0x` or `0X` are treated as hexadecimal values, and all other integers are treated as decimal values. Once you specify a stream's base, it processes all integers using that base until you specify a different one or until the program terminates.

Stream manipulator **`showbase`** causes octal numbers to be output with a leading `0` and hexadecimal numbers with either a leading `0x` or a leading `0X`—Section 19.6.10 shows that stream manipulator `uppercase` determines which option is chosen for hexadecimal values. Figure 19.14 demonstrates `showbase`. To reset the `showbase` setting, insert the stream manipulator **`noshowbase`** in the stream.

```
1  // fig19_14.cpp
2  // Stream manipulator showbase.
3  #include <iostream>
4
5  int main() {
6     int x{100};
7
8     // use showbase to show number base
9     std::cout << "Printing octal and hexadecimal values with showbase:\n"
10       << std::showbase;
11
12    std::cout << x << "\n"; // print decimal value
13    std::cout << std::oct << x << "\n"; // print octal value
14    std::cout << std::hex << x << "\n"; // print hexadecimal value
15 }
```

```
Printing octal and hexadecimal values with showbase:
100
0144
0x64
```

Fig. 19.14 Stream manipulator `showbase`.

19.6.9 Floating-Point Numbers; Scientific and Fixed Notation (`scientific`, `fixed`)

When you display a floating-point number without specifying its format, its value determines the output format. Some numbers display in scientific notation and others in fixed-point notation. The sticky stream manipulators `scientific` and `fixed` control the output format of floating-point numbers:

- **`scientific`** forces a floating-point number to display in scientific format.
- `fixed` forces a floating-point number to display in fixed-point notation with a specific number of digits to the right of the decimal point, as specified by member function `precision` or stream manipulator `setprecision`.

Figure 19.15 displays floating-point numbers in fixed and scientific formats using stream manipulators `scientific` (line 15) and `fixed` (line 19). The exponent format in scientific notation might vary among compilers.

```
1  // fig19_15.cpp
2  // Floating-point values displayed in system default,
3  // scientific and fixed formats.
4  #include <iostream>
5
6  int main() {
```

```
      double x{0.001234567};
      double y{1.946e9};

      // display x and y in default format
      std::cout << "Displayed in default format:\n" << x << '\t' << y;

      // display x and y in scientific format
      std::cout << "\n\nDisplayed in scientific format:\n"
         << std::scientific << x << '\t' << y;

      // display x and y in fixed format
      std::cout << "\n\nDisplayed in fixed format:\n"
         << std::fixed << x << '\t' << y << "\n";
   }
```

```
Displayed in default format:
0.00123457      1.946e+09

Displayed in scientific format:
1.234567e-03    1.946000e+09

Displayed in fixed format:
0.001235        1946000000.000000
```

Fig. 19.15 Floating-point values displayed in system default, scientific and fixed formats.

19.6.10 Uppercase/Lowercase Control (`uppercase`)

Stream manipulator **`uppercase`** outputs an uppercase `X` with hexadecimal-integer values or an uppercase `E` with scientific-notation floating-point values (Fig. 19.16; line 11). Using `uppercase` also displays the hexadecimal digits `A`–`F` in uppercase. These appear in lowercase by default. To reset the `uppercase` setting, output **`nouppercase`**.

```
1   // fig19_16.cpp
2   // Stream manipulator uppercase.
3   #include <iostream>
4
5   int main() {
6      std::cout << "Printing uppercase letters in scientific\n"
7         << "notation exponents and hexadecimal values:\n";
8
9      // use std::uppercase to display uppercase letters; use std::hex and
10     // std::showbase to display hexadecimal value and its base
11     std::cout << std::uppercase << 4.345e10 << "\n"
12        << std::hex << std::showbase << 123456789 << "\n";
13  }
```

```
Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+10
0X75BCD15
```

Fig. 19.16 Stream manipulator `uppercase`.

19.6.11 Specifying Boolean Format (`boolalpha`)

C++ `bool` values may be `false` or `true`. Recall that `0` also indicates `false`, and any nonzero value indicates `true`. A `bool` value displays as `0` or `1` by default. You can use stream manipulator **`boolalpha`** to set the output stream to display `bool` values as the strings `"true"` and `"false"` and stream manipulator **`noboolalpha`** to set the output stream back to displaying `bool` values as the integers `0` and `1`. Figure 19.17 demonstrates these stream manipulators. Line 9 displays `booleanValue` (which line 6 sets to `true`) as an integer. Line 13 uses `boolalpha` to display the `bool` value as a string. Lines 16–17 then change the `booleanValue` to `false` and use manipulator `noboolalpha`, so line 20 can display the `bool` value as an integer. Line 24 uses manipulator `boolalpha` to display the `bool` value as a string. Both `boolalpha` and `noboolalpha` are sticky settings.

```
1  // fig19_17.cpp
2  // Stream manipulators boolalpha and noboolalpha.
3  #include <iostream>
4
5  int main() {
6     bool booleanValue{true};
7
8     // display default true booleanValue
9     std::cout << "booleanValue is " << booleanValue;
10
11    // display booleanValue after using boolalpha
12    std::cout << "\nbooleanValue (after using boolalpha) is "
13       << std::boolalpha << booleanValue;
14
15    std::cout << "\n\nswitch booleanValue and use noboolalpha\n";
16    booleanValue = false; // change booleanValue
17    std::cout << std::noboolalpha; // use noboolalpha
18
19    // display default false booleanValue after using noboolalpha
20    std::cout << "\nbooleanValue is " << booleanValue;
21
22    // display booleanValue after using boolalpha again
23    std::cout << "\nbooleanValue (after using boolalpha) is "
24       << std::boolalpha << booleanValue << "\n";
25 }
```

```
booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false
```

Fig. 19.17 Stream manipulators `boolalpha` and `noboolalpha`.

19.6.12 Setting and Resetting the Format State via Member Function `flags`

20 Throughout Section 19.7, we've used stream manipulators to change output format characteristics. How do you return an output stream's format to its previous state after changing its

format? Member function **flags** without an argument returns the current **format state** settings as an **fmtflags** object. Member function flags with an fmtflags argument sets the format state as specified by the argument and returns the prior state settings. The initial settings of the value that flags returns might vary among compilers. Figure 19.18 uses member function flags to save the stream's original format state (line 16), then restore the original format settings (line 24). The capabilities for capturing the format state and restoring it are not required in C++20 text formatting. Each std::format call's formatting is used only in that call and does not affect any other std::format call, thus simplifying output formatting.

```
1   // fig19_18.cpp
2   // flags member function.
3   #include <format>
4   #include <iostream>
5
6   int main() {
7      int integerValue{1000};
8      double doubleValue{0.0947628};
9
10     // display flags value, int and double values (original format)
11     std::cout << std::format("flags value: {}\n", std::cout.flags())
12        << "int and double in original format:\n"
13        << integerValue << '\t' << doubleValue << "\n\n";
14
15     // save original format, then change the format
16     auto originalFormat{std::cout.flags()};
17     std::cout << std::showbase << std::oct << std::scientific;
18
19     // display flags value, int and double values (new format)
20     std::cout << std::format("flags value: {}\n", std::cout.flags())
21        << "int and double in new format:\n"
22        << integerValue << '\t' << doubleValue << "\n\n";
23
24     std::cout.flags(originalFormat); // restore format
25
26     // display flags value, int and double values (original format)
27     std::cout << std::format("flags value: {}\n", std::cout.flags())
28        << "int and double in original format:\n"
29        << integerValue << '\t' << doubleValue << "\n";
30  }
```

```
flags value: 513
int and double in original format:
1000    0.0947628

flags value: 5129
int and double in new format:
01750   9.476280e-02

flags value: 513
int and double in original format:
1000    0.0947628
```

Fig. 19.18 flags member function.

19.7 Stream Error States

Each stream object contains a set of **state bits** representing the stream's state—sticky format settings, error indicators, etc. You can use this information to test, for example, whether an input was successful. These state bits are defined in class `ios_base`—the base class of the stream classes. Stream extraction sets the stream's **`failbit`** to `true` if the wrong type of data is input. Similarly, stream extraction sets the stream's **`badbit`** to `true` if the operation fails in an unrecoverable manner—for example, if a disk fails when a program is reading a file from that disk. Figure 19.19 shows how to use bits like `failbit` and `badbit` to determine a stream's state.[4] Online Appendix E discusses bits and bit manipulation in detail.

---

4. The actual values output by this program may vary among compilers.

```
1  // fig19_19.cpp
2  // Testing error states.
3  #include <iostream>
4
5  int main() {
6     int integerValue{0};
7
8     // display results of cin functions
9     std::cout << std::boolalpha << "Before a bad input operation:"
10       << "\ncin.rdstate(): " << std::cin.rdstate()
11       << "\n  cin.eof(): " << std::cin.eof()
12       << "\n  cin.fail(): " << std::cin.fail()
13       << "\n  cin.bad(): " << std::cin.bad()
14       << "\n  cin.good(): " << std::cin.good()
15       << "\n\nExpects an integer, but enter a character: ";
16
17    std::cin >> integerValue; // enter character value
18
19    // display results of cin functions after bad input
20    std::cout << "\nAfter a bad input operation:"
21       << "\ncin.rdstate(): " << std::cin.rdstate()
22       << "\n  cin.eof(): " << std::cin.eof()
23       << "\n  cin.fail(): " << std::cin.fail()
24       << "\n  cin.bad(): " << std::cin.bad()
25       << "\n  cin.good(): " << std::cin.good();
26
27    std::cin.clear(); // clear stream
28
29    // display results of cin functions after clearing cin
30    std::cout << "\n\nAfter cin.clear()"
31       << "\ncin.fail(): " << std::cin.fail()
32       << "\ncin.good(): " << std::cin.good() << "\n";
33 }
```

```
Before a bad input operation:
cin.rdstate(): 0
    cin.eof(): false
```

```
   cin.fail(): false
    cin.bad(): false
   cin.good(): true

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
    cin.eof(): false
   cin.fail(): true
    cin.bad(): false
   cin.good(): false

After cin.clear()
cin.fail(): false
cin.good(): true
```

Fig. 19.19 Testing error states.

Member Function `eof`

The program begins by displaying the stream's state before receiving any input from the user (lines 9–14). Line 11 uses member function `eof` to determine whether end-of-file has been encountered on the stream. In this case, the function returns `0` (`false`). The function checks the value of the stream's **`eofbit`** data member, which is set to `true` for an input stream after end-of-file is encountered after an attempt to extract data beyond the end of the stream.

Member Function `fail`

Line 12 uses the **`fail`** member function to determine whether a stream operation has failed. The function checks the value of the stream's `failbit` data member, which is set to `true` on a stream when a format error occurs and as a result no characters are input. For example, this might occur when you attempt to read a number but the user enters a string. In this case, the function returns `0` (`false`). When such an error occurs on input, the characters are not lost. Usually, recovering from such input errors is possible.

Member Function `bad`

Line 13 uses the **`bad`** member function to determine whether a stream operation failed. The function checks the value of the stream's `badbit` data member, which is set to `true` for a stream when an error occurs that results in the loss of data—such as reading from a file when the disk on which the file is stored fails. In this case, the function returns 0 (`false`). Generally, such serious failures are nonrecoverable.

Member Function `good`

Line 14 uses the **`good`** member function, which returns `true` if the `bad`, `fail` and `eof` functions would all return `false`. The function checks the stream's **`goodbit`**, which is set to `true` for a stream if none of the bits `eofbit`, `failbit` or `badbit` is set to `true` for the stream. In this case, the function returns `1` (`true`). I/O operations should be performed only on "good" streams.

Member Function `rdstate`

The **`rdstate`** member function (line 10) returns the stream's overall error state as an integer value. The function's return value could be tested, for example, by a `switch` statement that

examines `eofbit`, `badbit`, `failbit` and `goodbit`. The preferred means of testing the state of a stream is to use member functions `eof`, `bad`, `fail` and `good`—using these functions does not require you to be familiar with particular status bits.

Causing an Error in the Input Stream and Redisplaying the Stream's State

Line 17 reads a value into an `int` variable. Enter a string rather than an `int` to force an error to occur in the input stream. At this point, the input fails and lines 20–25 once again call the streams's state functions. In this case, `fail` returns `1` (`true`) because the input failed. Function `rdstate` also returns a nonzero value (`true`) because at least one of the member functions `eof`, `bad` and `fail` returned `true`. Once an error occurs in the stream, function `good` returns `0` (`false`).

Clearing the Error State, So You May Continue Using the Stream

After an error occurs, you can no longer use the stream until you reset its error state.
The `clear` member function (line 27) is used to restore a stream's state to "good" so that I/O may proceed on that stream. Lines 30–32 show that `fail` returns `0` (`false`) and `good` returns `1` (`true`), so the input stream can be used again.

The default argument for `clear` is `goodbit`, so the statement

```
std::cin.clear();
```

clears `cin` and sets `goodbit` for the stream. The statement

```
std::cin.clear(ios::failbit)
```

sets the `failbit`. You might want to do this when performing input on `cin` with a user-defined type and encountering a problem. The name `clear` might seem inappropriate in this context, but it's correct.

Overloaded Operators `!` and `bool`

11 Overloaded operators can be used to test a stream's state in conditions. The `operator!` member function—inherited into the stream classes from class `basic_ios`—returns `true` if the `badbit`, the `failbit` or both are `true`. The `operator bool` member function (added in C++11) returns `false` if the `badbit` is `true`, the `failbit` is `true` or both are `true`. These functions are useful in I/O processing when a `true`/`false` condition is being tested under the control of a selection statement or iteration statement. For example, you could use an `if` statement of the form

```
if (!std::cin) {
   // process invalid input stream
}
```

to execute code if `cin`'s stream is invalid due to a failed input. Similarly, you've already seen a `while` condition of the form

```
while (std::cin >> variableName) {
   // process valid input
}
```

which enables the loop to execute as long as each input operation is successful and terminates the loop if an input fails or the end-of-file indicator is encountered.

19.8 Tying an Output Stream to an Input Stream

Interactive command-line applications generally use an `istream` for input and an `ostream` for output. When a prompting message appears on the screen, the user responds by entering the appropriate data. Obviously, the prompt needs to appear before the input operation proceeds. With output buffering, outputs appear only

- when the buffer fills
- when outputs are flushed explicitly by the program or
- automatically at the end of the program.

C++ provides member function **`tie`** to synchronize (i.e., "tie together") an `istream` and an `ostream` to ensure that outputs happen (that is, they are flushed) before subsequent inputs. The call

```
std::cin.tie(&cout);
```

ties `cout` (an `ostream`) to `cin` (an `istream`). This particular call is redundant because C++ performs this operation automatically for the standard output and input streams. However, you might use this with other input/output stream pairs. To untie an input stream, `inputStream`, from an output stream, use the call

```
inputStream.tie(0);
```

20 19.9 C++20 Text Formatting

As we mentioned in Chapter 3, C++20 provides powerful string-formatting capabilities via the **`format` function** (in header **`<format>`**). These capabilities greatly simplify formatting by using a concise syntax that's based on the Python programming language's text formatting. As you've seen in this chapter, pre-C++20 output formatting is quite verbose. C++20 text-formatting capabilities are more concise and more powerful.

20 We presented several new C++20 text-formatting features throughout the book using the open-source `{fmt}` library's `fmt::format` function because our preferred compilers did not yet support the C++20 `<format>` header and `std::format` function. As we wrote this online chapter, Microsoft Visual C++ added support for these, so we used the `<format>` header and `std::format` function for this section.

20 19.9.1 C++20 `std::format` Presentation Types

When you specify a placeholder for a value in a format string, the `std::format` function assumes the value should be displayed as a string unless you specify another type. In some cases, the type is required—for example, if you want to specify precision for a floating-point number or change the base in which to display an integer. In these cases, you can specify the **presentation type** in each placeholder. Figure 19.20 shows the various presentation types.

```
1  // fig19_20.cpp
2  // C++20 text-formatting presentation types.
3  #include <format>
4  #include <iostream>
5
6  int main() {
```

```
 7      // floating-point presentation types
 8      std::cout << "Display 17.489 with default, .1 and .2 precisions:\n"
 9         << std::format("f:{0:f}\n.1f: {0:.1f}\n.2f: {0:.2f}\n\n", 17.489);
10
11      std::cout << "Display 10000000000000000.0 with f, e, g and a\n"
12         << std::format("f: {0:f}\ne: {0:e}\ng: {0:g}\na: {0:a}\n\n",
13               10000000000000000.0);
14
15      // integer presentation types; # displays a base prefix
16      std::cout << "Display 100 with d, #b, #o and #x:\n"
17         << std::format(
18              "d: {0:d}\n#b: {0:#b}\n#o: {0:#o}\n#x: {0:#x}\n\n", 100);
19
20      // character presentation type
21      std::cout << "Display 65 and 97 with c:\n"
22         << std::format("{:c} {:c}\n\n", 65, 97);
23
24      // string presentation type
25      std::cout << "Display \"hello\" with s:\n"
26         << std::format("{:s}\n", "hello");
27   }
```

```
Display 17.489 with default, .1 and .2 precisions:
f: 17.489000
.1f: 17.5
.2f: 17.49

Display 10000000000000000.0 with f, e, g and a
f: 10000000000000000.000000
e: 1.000000e+16
g: 1e+16
a: 1.1c37937e08p+53

Display 100 with d, #b, #o and #x:
d: 100
#b: 0b1100100
#o: 0144
#x: 0x64

Display 65 and 97 with c:
A a

Display "hello" with s:
hello
```

Fig. 19.20 C++20 text-formatting presentation types.

20 Floating-Point Values and C++20 Presentation Type `f`

You've used the **presentation type** `f` to format floating-point values. Formatting is **type-dependent**, so this presentation type is required to specify a floating-point number's precision. Line 9 uses the `f` presentation type to format the `double` value `17.489` in the default precision (`6`) and rounded to the tenths and hundredths positions. Function `std::format` uses presentation types to determine whether the other formatting options are allowed for a given type. For all the presentation types, their formatting options and the order in which the options must be specified, see

```
https://en.cppreference.com/w/cpp/utility/format/formatter
```

Indexing Arguments By Position

In line 9, note the `0` to the left of each format specifier's colon (:). You can reference the arguments after the format string **positionally** by their index numbers starting from index `0`. This allows you to

- **reference the same argument multiple times**, as we did three times in line 9, or
- **reference the arguments in any order**, which can be helpful when localizing applications for spoken languages that order words differently in sentences.

20 Floating-Point Values and C++20 Presentation Types `e`, `g` and `a`

You also can format floating-point values using the following presentation types, as shown in lines 12–13:

- `e` or `E`—These use **exponential (scientific) notation** to format floating-point values. The exponent is preceded by an `e` or `E` in the formatted string. The value `1.000000e+16` in this program's output is equivalent to

```
1.000000 x 10^16
```

- `g or G`—These choose between fixed-point notation and exponential notation based on the value's magnitude. For exponential notation, `g` displays a lowercase `e`, and `G` displays an uppercase `E`.
- `a` **or** `A`—These format floating-point values in hexadecimal notation with lowercase letters (`a`) or uppercase letters (`A`), respectively.

20 C++20 Integer Presentation Types

Lines 17–18 display the `int` value `100` using various integer number systems:[5]

---

5. See the online Appendix C, Number Systems for information about the binary, octal and hexadecimal number systems.

- `d`—Displays an integer in decimal (base 10) format.
- `b` **or** `B`—These display an integer in binary (base 2) format with a lowercase `b` or uppercase `B` when a binary value is displayed with its base (Section 19.9.3).
- `o` **presentation type**—Displays an integer in octal (base 8) format.
- `x` **or** `X` **presentation type**—These display an integer in hexadecimal (base 16) format with a lowercase or uppercase letters, respectively.

20 C++20 Character Presentation Type

The `c` **presentation type** formats an integer character code as the corresponding character, as shown in line 22.

20 C++20 String Presentation Type

When a placeholder does not specify a presentation type, the default is to format the corresponding value as a string. The `s` **presentation type** indicates that the corresponding value must specifically be a string, an expression that produces a string or a string literal, as in line 26.

20 C++20 Locale-Specific Numeric and `bool` Formatting

If your application requires locale-specific formatting of numeric or `bool` values, precede the integer or floating-point presentation type with `L`.

20 19.9.2 C++20 `std::format` Field Widths and Alignment

Previously you used **field widths** to format text in a specified number of character positions. Figure 19.21 demonstrates field widths, default alignments and explicit alignments. We enclose each formatted value in brackets (`[]`) so you can better see the formatting results. By default, `std::format` **right-aligns numbers** and **left-aligns strings**, as demonstrated by line 8. For values with fewer characters than the field width, the remaining character positions are filled with spaces.[6] Values with more characters than the specified field width use as many character positions as they need.

---

20 6. C++20 allows you to specify any fill character (other than `{` and `}`, which delimit placeholders) immediately to the right of the format specifier's colon (`:`).

```
1   // fig19_21.cpp
2   // C++20 text-formatting with field widths and alignment.
3   #include <format>
4   #include <iostream>
5
6   int main() {
7      std::cout << "Default alignment with field width 10:\n"
8         << std::format("[{:10d}]\n[{:10f}]\n[{:10}]\n\n", 27, 3.5, "hello");
9
10     std::cout << "Specifying left or right alignment in a field:\n"
11        << std::format("[{:<15d}]\n[{:<15f}]\n[{:>15}]\n\n",
12              27, 3.5, "hello");
13
14     std::cout << "Centering text in a field:\n"
15        << std::format("[{:^7d}]\n[{:^7.1f}]\n[{:^7}]\n", 27, 3.5, "hello");
16  }
```

```
Default alignment with field width 10:
[        27]
[  3.500000]
[hello     ]

Specifying left or right alignment in a field:
[27             ]
[3.500000       ]
[          hello]

Centering text in a field:
[  27   ]
[  3.5  ]
[ hello ]
```

Fig. 19.21 C++20 text-formatting with field widths and alignment.

Recall that you can specify left-alignment and right-alignment with < and >. Lines 11–12 **left-align** the numeric values `27` and `3.5` and **right-align** the string `"hello"` in fields of 15 characters

The I/O streams output formatting shown earlier in this chapter does not support **center-aligning text**, but `std::format` can do this conveniently with ^, as shown in line 15. Centering attempts to spread the unoccupied character positions equally to the left and right of the formatted value. `std::format` places the extra space to the right if an odd number of character positions remain, as you can see for the value `27`, which has two spaces to its left and three to its right.

20 19.9.3 C++20 `std::format` Numeric Formatting

Figure 19.22 demonstrates various C++20 numeric formatting capabilities. By default, negative numeric values display with a `-` sign. Sometimes it's desirable to force a `+` sign to display for a positive number. A `+` in the format specifier (line 8) indicates that the numeric value should always be preceded by a sign (`+` or `-`). To fill the field's remaining characters with `0`s rather than spaces, place `0` before the field width and *after* the `+` if there is one, as in line 8's second format specifier. A space in the format specifier (as in line 11's second and third format specifiers) indicates that positive numbers should show a space character in the sign position. This is useful for aligning positive and negative values for display purposes. Note that the two values with a space in their format specifiers align. If a field width is specified, place the space before the field width. To precede a binary, octal or hexadecimal number with its base, use `#` in the format specifier as in line 14.

```
1   // fig19_22.cpp
2   // C++20 text-formatting numeric formatting options.
3   #include <format>
4   #include <iostream>
5
6   int main() {
7      std::cout << "Displaying signs and padding with leading 0s:\n"
8         << std::format("[{0:+10d}]\n[{0:+010d}]\n\n", 27);
9
10     std::cout << "Displaying a space before a positive value:\n"
11        << std::format("{0:d}\n{0: d}\n{1: d}\n\n", 27, -27);
12
13     std::cout << "Displaying a base indicator before a number:\n"
14        << std::format("{0:d}\n{0:#b}\n{0:#o}\n{0:#x}\n", 100);
15  }
```

```
Displaying signs and padding with leading 0s:
[       +27]
[+000000027]

Displaying a space before a positive value:
27
 27
-27

Displaying a base indicator before a number:
100
0b1100100
```

```
0144
0x64
```

Fig. 19.22 C++20 text-formatting numeric formatting options.

20 19.9.4 C++20 `std::format` Field Width and Precision Placeholders

You can programmatically specify field widths and precisions using **nested placeholders** in a format specifier. Figure 19.23 displays the `double` value `123.456` in a field of `8` characters with precisions of `0`–`4`. In line 13, the argument `value` is the number to format. In the format specifier `"{:{}.{}f}"`, the nested placeholders to the right of the colon (`:`) are replaced left-to-right by the values of the arguments `width` and `precision`, respectively.

```
 1   // fig19_23.cpp
 2   // C++20 text-formatting field width and precision placeholders.
 3   #include <format>
 4   #include <iostream>
 5
 6   int main() {
 7      std::cout << "Demonstrating field width and precision placeholders:\n";
 8
 9      double value{123.456};
10      int width{8};
11
12      for (int precision{0}; precision < 5; ++precision) {
13         std::cout << std::format("{:{}.{}f}\n", value, width, precision);
14      }
15   }
```

```
Demonstrating field width and precision placeholders:
     123
   123.5
  123.46
 123.456
123.4560
```

Fig. 19.23 C++20 text-formatting field width and precision placeholders.

19.10 Wrap-Up

This chapter showed C++ input/output formatting with streams. You learned about the stream-I/O classes and predefined objects. We discussed `ostream`'s formatted and unformatted output capabilities performed by the `put` and `write` functions. You learned about `istream`'s formatted and unformatted input capabilities performed by the `eof`, `get`, `get-line`, `peek`, `putback`, `ignore` and `read` functions. We discussed stream manipulators and member functions that perform formatting tasks:

- `dec`, `oct`, `hex` and `setbase` for displaying integers
- `precision` and `setprecision` for controlling floating-point precision
- and `width` and `setw` for setting field width.

You also learned additional formatting with `iostream` manipulators and member functions:

- `showpoint` for displaying decimal point and trailing zeros
- `left`, `right` and `internal` for alignment
- `fill` and `setfill` for padding
- `scientific` and `fixed` for displaying floating-point numbers in scientific and fixed notation

- `uppercase` for uppercase/lowercase control
- `boolalpha` for specifying Boolean format
- and `flags` and `fmtflags` for resetting the format state.

You'll encounter many of the preceding capabilities in legacy C++ code.

Finally, we presented many of C++20's more concise and convenient text-formatting capabilities, including presentation types to specify data types to format, positional arguments, field widths, alignment, numeric formatting and using placeholders to specify field widths and precisions.

20. Other Topics and a Look Toward the Future of C++

Objectives

In this chapter, you'll:

- Use `const_cast` to temporarily treat a `const` object as a non-`const` object.
- Understand storage classes and storage duration.
- Use `namespaces` to ensure that identifiers are unique.
- Use `mutable` members in `const` objects.
- Use operator keywords in place of corresponding operator symbols.
- Use class-member pointer operators `.*` and `->*`.
- Determine an object's type with runtime type information (RTTI), `dynamic_cast`, `typeid` and `type_info`.
- Use the C++17 and C++20 `[[nodiscard]]` attribute to indicate that a function's return value should not be ignored.
- Using smart pointers to manage dynamic memory for objects shared throughout a program.
- Determine types at compile time with `decltype`.
- Receive braced initializers as function arguments.
- Use C++20 attributes `[[likely]]` and `[[unlikely]]` to specify selection-statement paths that are likely or unlikely to execute, so the compiler can optimize accordingly.
- Get an overview of new features being considered for C++23.

Outline

20.1 Introduction
20.2 `const_cast` Operator
20.3 Storage Classes and Storage Duration
20.3.1 Storage Duration
20.3.2 Local Variables and Automatic Storage Duration
20.3.3 Static Storage Duration
20.3.4 `mutable` Class Members
20.3.5 Mechanical Demonstration of a `mutable` Data Member
20.4 `namespaces`
20.4.1 Defining `namespaces`
20.4.2 Accessing `namespace` Members with Qualified Names
20.4.3 `using` Directives Should Not Be Placed in Headers
20.4.4 Aliases for `namespace` Names
20.4.5 C++17 Nested Namespaces
20.5 Operator Keywords
20.6 Pointers to Class Members (`.*` and `->*`)
20.7 Case Study: Payroll System Using Polymorphism and Runtime Type Information — Downcasting, `dynamic_cast`, `typeid` and `type_info`
20.8 Inheriting Base-Class Constructors
20.9 C++17 and C++20: `[[nodiscard]]` Attribute
20.10 `shared_ptr` and `weak_ptr` Smart Pointers
20.10.1 Reference Counted `shared_ptr`
20.10.2 `weak_ptr`: `shared_ptr` Observer

20.11 Trailing Return Types for Functions
20.12 `decltype`
20.13 `initializer_list` Class Template
20.14 C++20: `[[likely]]` and `[[unlikely]]` Attributes
20.15 A Look Toward C++23
20.16 Wrap-Up

20.1 Introduction

20 23 This chapter presents miscellaneous C++ features. We'll occasionally update it online as we write about additional C++20 features and new features being developed for C++23 and beyond. The chapter discusses:

- The **`const_cast` operator**, which allows you to remove the `const` qualification of a variable.
- **Storage classes and storage duration**, which determine an object's lifetime.
- **`namespace`s**, which help ensure that every identifier in a program has a unique name, helping avoid name conflicts (for example, between your code and library code).
- **Operator keywords** for programmers who have keyboards that do not support certain characters used in operator symbols, such as `!`, `&`, `^`, `~` and `|`.
- **Operators `.*` and `->*` that you can use with pointers to class members** to access a data member or member function.
- **Navigating a class hierarchy** with **downcasting**.
- **Runtime type information (RTTI)**, which enables you to dynamically discover an object's type.
- **Inheriting base-class constructors**—useful when a derived class's constructors have the same parameters and simply pass the arguments to the base class's constructors.
- 17 **C++17's `[[nodiscard]]` attribute**, which indicates that a function's return value should not be ignored so compilers can warn you when the return value is not used in your program.
- 20 **C++20's `[[nodiscard]]` enhancement**, which allows you to specify a string that the compiler will display as the reason the function's caller should not ignore the return value.
- 11 **`shared_ptr` and `weak_ptr` smart pointers** for managing shared dynamically allocated objects.
- **Trailing return types for functions**, which are typically used in complex function templates.
- 11 **`decltype`**, which can be used at compile time to determine types—typically used in template metaprogramming.
- **Functions that receive `initializer_list`s**, enabling functions to process any number of arguments.
- 20 Attributes `[[likely]]` and `[[unlikely]]`, which specify branches of `if`...`else` or `switch` statements that are more or less likely to execute, so the compiler can optimize the code accordingly.
- 23 Features being considered for **C++23**.

20.2 `const_cast` Operator

SE C++ provides the **`const_cast`** operator[1] for casting away `const`. In general, it's dangerous to use the `const_cast` operator because it allows a program to modify a variable that was declared `const`. But, there are cases where it's desirable, or even necessary, to cast away `const`-ness. For example, older C and C++ libraries might provide functions that have non-`const` parameters and that do not modify their parameters—if you wish to pass `const` data to such a function, you'd need to cast away the

data's `const`-ness; otherwise, the compiler would report errors. Similarly, you could pass **non-`const` data** to a function that treats it as if it were constant then returns that data as a constant. In such cases, you might need to cast away the `const`-ness of the returned data, as we demonstrate in Fig. 20.1.

1. "`const_cast` conversion." Accessed March 11, 2022. https://en.cppreference.com/w/cpp/language/const_cast.

```
1  // fig20_01.cpp
2  // Demonstrating const_cast.
3  #include <cctype> // contains prototype for function toupper
4  #include <iostream>
5  #include <string>
6
7  // returns the larger of two strings
8  const std::string& maximum(
9     const std::string& first, const std::string& second) {
10    return (first > second ? first : second);
11 }
12
13 int main() {
14    std::string s1{"hello"}; // non-const string
15    std::string s2{"goodbye"}; // non-const string
16
17    // const_cast required to allow the const reference returned by
18    // maximum to be assigned to the std::string reference max
19    std::string &max{const_cast<std::string&>(maximum(s1, s2))};
20
21    std::cout << "The larger string is: " << max << "\n";
22
23    for (char& character : max) {
24       character = std::toupper(character);
25    }
26
27    std::cout << "The larger string capitalized is: " << max << "\n";
28 }
```

```
The larger string is: hello
The larger string capitalized is: HELLO
```

Fig. 20.1 Demonstrating operator `const_cast`.

In this program, function `maximum` (lines 8–11) receives two strings as `const std::string&` parameters and returns a `const std::string&` that refers to the larger string. Function `main` declares the two `std::string`s as non-`const` objects (lines 14–15); thus, **these strings are modifiable**. In `main`, we wish to output the larger of the two strings, then modify that string by converting it to uppercase letters.

Err Function `maximum`'s two parameters are of type `const std::string&`. If the function were to return `std::string&`, the compiler would issue an error message indicating that a `const std::string&` cannot be converted to a `std::string&`. If allowed, that would be dangerous because it attempts to treat data that the function believes to be `const` as if it were non-`const` data.

Though `maximum` believes the data to be constant, the original strings in `main` are not constant. Therefore, `main` should be able to modify the contents of those strings as necessary. We know these strings are modifiable, so we use `const_cast` (line 19) to demonstrate the mechanics of casting away the `const`-ness of the reference `maximum` returns. Lines 23– 25 then use that reference to convert the

contents of the larger string to uppercase letters. Without the `const_cast` in line 19, this program will not compile, because you are not allowed to assign a `const std::string&` to a `std::string&`. On the other hand, you can always assign a `std::string&` to a `const std::string&` so you can treat a non-`const` string as a constant.

CG Err **The C++ Core Guidelines say not to cast away `const`-ness.**[2] For cases in which you need `const_cast` in custom classes, such as sharing code between `const` and non-`const` member functions, **core guideline ES.50** provides sample code showing how to encapsulate this in the class definition. In general, a `const_cast` should be used only if you know that the original data is not constant—such as in a non-`const` member function of a class, which can be called only on a non-`const` object of the class.[3] Otherwise, unexpected results may occur.

2. C++ Core Guidelines, "ES.50: Don't cast away `const`." Accessed March 10, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const.
3. C++ Core Guidelines, "ES.50: Don't cast away `const`." Accessed March 10, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const.

20.3 Storage Classes and Storage Duration

As you know, programs use identifiers for variable names, functions and class names. The attributes of variables include **name**, **type**, **size** and **value**. Each identifier in a program also has other attributes, including scope, **linkage** and **storage duration**.

20 As we discussed in Section 5.10, an identifier's scope is where the identifier can be referenced in a program. Some identifiers can be referenced throughout a program; others can be referenced from only limited portions of a program. An identifier's linkage determines whether it's known only in the source file where it's declared or across multiple files that are compiled, then linked together. C++20 modules (Chapter 16) introduce **module linkage** for identifiers known only in the module that defines them. An identifier's storage-class specifier helps determine its storage duration and linkage.

Storage Class Specifiers

C++ provides several **storage-class specifiers** that determine a variable's storage duration: **`extern`**, **`mutable`**, **`static`** and `thread_local`. Storage-class specifier `mutable` is used exclusively with classes, and `thread_local` is used in multithreaded applications.

20.3.1 Storage Duration

An identifier's **storage duration** determines the period during which that identifier's storage exists in memory.[4] Some exist briefly, some are repeatedly created and destroyed, and others exist for a program's entire execution.

4. "Storage class specifiers." Accessed March 11, 2022. https://en.cppreference.com/w/cpp/language/storage_duration.

The storage-class specifiers can be split into four storage durations: **automatic**, **static**, **dynamic** and **thread**. In Chapter 11, you learned that you can dynamically allocate additional memory at execution time. Variables allocated dynamically have **dynamic storage duration**. The rest of this section focuses on automatic and static storage duration, and `mutable` data members.

20.3.2 Local Variables and Automatic Storage Duration

Variables with **automatic storage duration** include:

- local variables declared in functions
- function parameters

Such automatic variables are created when program execution enters the block in which they're defined. They exist while the block is active, and they're destroyed when the program exits the block. An automatic variable exists only from where it's defined to the closing brace of the block in which the definition appears, or for the entire function body in the case of a function parameter. Local variables are of automatic storage duration by default.

Perf Automatic storage is a means of conserving memory because automatic storage duration variables exist in memory only when the block in which they're defined is executing. Declare variables as close as possible to where they're first used.

20.3.3 Static Storage Duration

Keywords `extern` and `static` declare identifiers for variables with **static storage duration** and functions. Variables with static storage duration exist in memory from the point at which the program begins execution until the program terminates. Such a variable is initialized once when its declaration is encountered. A function name exists when the program begins execution. Even though function names and static-storage-duration variables exist from the start of program execution, their scope determines where they can be used in the program.

Identifiers with Static Storage Duration

20 There are two types of identifiers with static storage duration—external identifiers (such as global variables) and local variables declared with the storage-class specifier `static`. **Global variables** are created by placing variable declarations outside any class or function definition. Global variables retain their values throughout a program's execution. Global variables and global functions can be referenced by any function that follows their declarations or definitions in the source file. If a global variable is declared `static`, it is known only in that translation unit. If a C++20 module (Chapter 16) contains a global `static` variable, that variable may not be exported by the module. In general, you should avoid global variables.

`static` Local Variables

Local `static` variables are known only in the function that declares them but retain their values when the function returns to its caller. The next time the function is called, each `static` local variable contains the value it had when the function last completed execution. The following statement declares local `static` variable `count` and initializes it to 1:

```
static int count{1};
```

The initialization occurs only the first time this statement is encountered. All numeric variables of `static` storage duration are **initialized to zero by default**. It's nevertheless a good practice to explicitly initialize all variables, so each variable's initial value is clear.

Storage-class specifiers `extern` and `static` determine an identifier's **linkage** when they're applied explicitly to external identifiers such as global variables and global function names:[5]

- `extern` indicates that the identifier is visible to other translation units.
- `static` indicates that the identifier is visible only in the current translation unit.

5. "Storage class specifiers." Accessed March 11, 2022. https://en.cppreference.com/w/cpp/language/storage_duration.

20 C++20 modules (Chapter 16) must explicitly `export` identifiers to make them visible to other translation units.

20.3.4 `mutable` Class Members

Section 20.2 introduced the `const_cast` operator for removing the "`const`-ness" of a type. A `const_cast` operation can also be applied to a data member of a `const` object from the body of a `const` member function of that object's class. This enables the `const` member function to modify the data member, even though the member function considers the object to be `const`.

For example, consider a linked list that maintains its contents in sorted order. Searching through the linked list does not require modifications to the data of the linked list, so the search function could be a `const` member function of the linked-list class. However, it's conceivable that a linked-list object—to make subsequent searches more efficient— might keep track of the location of the last successful match. If the next search operation attempts to locate an item that appears later in the list, the search could begin from the location of the last successful match rather than from the beginning of the list. To do this, the `const` member function that performs the search must be able to modify the data member that keeps track of the last successful search.

SE CG For a class with a "secret" implementation detail that should always be modifiable— such as our linked-list example—the C++ Core Guidelines recommend using the storage-class specifier **`mutable`** to designate that a data member is always modifiable, even in a `const` member function or `const` object.[6,7] Though `mutable` is a storage-class specifier, it does not affect a variable's storage duration or linkage.

6. C++ Core Guidelines, "ES.50: Don't cast away `const`." Accessed March 10, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-casts-const.
7. "cv (`const` and `volatile`) type qualifiers—`mutable` specifier." Accessed March 11, 2022. https://en.cppreference.com/w/cpp/language/cv.

20.3.5 Mechanical Demonstration of a `mutable` Data Member

Figure 20.2 presents a mechanical example of a `mutable` member to prove that such a member is always modifiable. Class `TestMutable` (lines 6–15) contains

- a constructor (line 8),
- function `getValue` (lines 10–12), which is a `const` member function that returns a copy of `value`, and
- a `private` data member `value` that's declared `mutable` (line 14).

Function `getValue` increments the `mutable` data member `value` in the `return` statement (line 11). Typically, a `const` member function cannot modify the object on which it's called. Because the data member `value` is `mutable`, this `const` function can modify the data.

```
1  // fig20_02.cpp
```

```
// Demonstrating storage-class specifier mutable.
#include <iostream>

// class TestMutable definition
class TestMutable {
public:
   TestMutable(int v = 0) : value{v} { }

   int getValue() const {
      return ++value; // increments value
   }
private:
   mutable int value; // mutable member
};

int main() {
   const TestMutable test{99};

   std::cout << "Initial value: " << test.getValue()
      << "\nModified value: " << test.getValue() << "\n";
}
```

```
Initial value: 100
Modified value: 101
```

Fig. 20.2 Demonstrating a `mutable` data member.

Line 18 declares `const TestMutable` object `test` and initializes it to `99`. Line 20 calls the `const` member function `getValue`, which adds one to `value` and returns its new value. The compiler allows the `getValue` call on the object `test` because it's a `const` object and `getValue` is a `const` member function. However, `getValue` modifies variable `value`. Thus, when line 20 invokes `getValue`, the new `value` (`100`) is output to prove that the `mutable` data member was modified. Line 21 demonstrates this again, displaying `101`.

20.4 namespaces

A program may include many identifiers defined in different scopes. Sometimes a variable of one scope will collide with a variable of the same name in a different scope, possibly creating a naming conflict. Such overlapping can occur at many levels. Identifier overlapping frequently occurs in libraries that use the same names for global identifiers (such as functions). This can cause compilation errors for multiple definitions of the same identifiers.

C++ solves this problem with **namespaces**.[8] Each `namespace` defines a scope containing identifiers. To use a **`namespace` member**, the member's name must be qualified with the `namespace` name and the **scope resolution operator (`::`)**, as in

8. "Namespaces." Accessed March 11, 2022. https://en.cppreference.com/w/cpp/language/namespace.

```
MyNameSpace::member
```

or a `using` directive must appear before the name is used in the program. Typically, such `using` statements are placed at the beginning of the file that uses the `namespace` members. For example, placing the following `using` directive at the beginning of a source-code file

```
using namespace MyNameSpace;
```

specifies that the code in the file can use *MyNameSpace* members without preceding each member with

```
MyNameSpace::
```

A `using` declaration of the form

```
using std::cout;
```

brings one name into the scope where the declaration appears. A `using` directive of the following form brings all the names from the specified `namespace` (`std`) into the scope where the directive appears:

```
using namespace std;
```

SE 20 Generally, you should precede a member with its namespace name and the scope resolution operator (`::`) to prevent naming conflicts. Not all `namespace`s are guaranteed to be unique. Two third-party vendors might inadvertently use the same identifiers for their `namespace` names. Figure 20.3 demonstrates `namespace`s. Namespaces are commonly used in Modern C++, and now in **C++20 modules**. See Chapter 16 for more information.

```
// fig20_03.cpp
// Demonstrating namespaces.
#include <fmt/format.h>
#include <iostream>

int integer1{98}; // global variable

// create namespace Example
namespace Example {
   // declare two constants and one variable
   const double pi{3.14159};
   const double e{2.71828};
   int integer1{8};

   void printValues(); // prototype

   // nested namespace
   namespace Inner {
      // define enumeration
      enum Years {fiscal1 = 2020, fiscal2, fiscal3};
   }
}

// create unnamed namespace
namespace {
   double doubleInUnnamed{88.22}; // declare variable
}

int main() {
   // output value doubleInUnnamed of unnamed namespace
```

```
31      std::cout << fmt::format("doubleInUnnamed = {}\n", doubleInUnnamed);
32
33      // output global variable
34      std::cout << fmt::format("(global) integer1 = {}\n", integer1);
35
36      // output values of Example namespace
37      std::cout << fmt::format(
38         "pi = {}\ne = {}\ninteger1 = {}\nfiscal3 = {}\n", Example::pi,
39         Example::e, Example::integer1, Example::Inner::fiscal3);
40
41      Example::printValues(); // invoke printValues function
42   }
43
44   // display variable and constant values
45   void Example::printValues() {
46      std::cout << "\nIn printValues:\n";
47      std::cout << fmt::format(
48         "integer1 = {}\npi = {}\ne = {}\n",  pi, e, integer1);
49      std::cout << fmt::format(
50         "doubleInUnnamed = {}\n(global) integer1 = {}\nfiscal3 = {}\n",
51         doubleInUnnamed, ::integer1, Inner::fiscal3);
52   }
```

```
doubleInUnnamed = 88.22
(global) integer1 = 98
pi = 3.14159
e = 2.71828
integer1 = 8
fiscal3 = 2022

In printValues:
integer1 = 8
pi = 3.14159
e = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
fiscal3 = 2022
```

Fig. 20.3 Demonstrating the use of `namespaces`.

20.4.1 Defining `namespaces`

Err Lines 9–22 use the keyword `namespace` to define `namespace` `Example`, which like a class, has a body delimited by braces (`{}`). `Example`'s members consist of two constants (`pi` and `e` in lines 11–12), an `int` (`integer1` in line 13), a function (`printValues` in line 15) and a **nested namespace** (`Inner` in lines 18–21). `Example`'s `integer1` member has the same name as global variable `integer1` (line 6). Variables with the same name must have different scopes; otherwise, compilation errors occur. A `namespace` can contain constants, data, classes, nested `namespaces`, functions, etc. Definitions of `namespaces` must occupy the global scope or be nested within other `namespaces`. Unlike classes, different `namespace` members can be defined in separate `namespace` blocks. For example, each standard library header has a `namespace` block placing its contents in `namespace` `std`.

Lines 25–27 create an **unnamed namespace** containing `doubleInUnnamed`. Variables, classes and functions in an unnamed `namespace` are accessible only in the current translation unit. However, unlike variables, classes or functions with `static` linkage, those in the unnamed `namespace` may be used

as template arguments. The unnamed `namespace` has an implicit `using` directive, so its members appear to occupy the **global namespace**. They are accessible directly and do not have to be qualified with a `namespace` name. Global variables are also part of the global `namespace` and are accessible in all scopes following the declaration in the file. Each separate translation unit has its own unique unnamed `namespace`.

20.4.2 Accessing `namespace` Members with Qualified Names

Line 31 outputs the value of variable `doubleInUnnamed`, which is directly accessible as part of the unnamed `namespace`. Line 34 outputs the value of global variable `integer1`. For both of these variables, the compiler first attempts to find a local declaration of the variables in `main`. There are no local declarations, so the compiler assumes those variables are in the global `namespace`.

Lines 37–39 output the values of `pi`, `e`, `integer1` and `fiscal3` from the `Example namespace`. Each must be qualified with `Example::` because the program does not provide any `using` directive or `using` declarations indicating that it will use `Example`'s members. In addition, member `integer1` must be qualified because a global variable has the same name. Otherwise, the global variable's value is output. `fiscal3` is a member of nested `namespace Inner`, so it must be qualified with `Example::Inner::`.

Function `printValues` (defined in lines 45–52) is in the `Example namespace`, so it can access other `Example` members directly without using a `namespace` qualifier. Line 47–51 output `integer1`, `pi`, `e`, `doubleInUnnamed`, global variable `integer1` and `fiscal3`. Notice that `pi` and `e` are not qualified with `Example`. Variable `doubleInUnnamed` is still accessible because it's in the unnamed `namespace`, and the variable name does not conflict with any other `Example` members. The global version of `integer1` must be qualified with the scope resolution operator (`::`) because its name conflicts with an `Example` member. Also, `fiscal3` must be qualified with `Inner::`. When accessing members of a nested `namespace`, the members must be qualified with the `namespace` name unless the member is used within the nested `namespace`. Placing `main` in a `namespace` is a compilation error.

20.4.3 `using` Directives Should Not Be Placed in Headers

`namespace`s are particularly useful in large-scale applications that use many class libraries. In such cases, there's a higher likelihood of naming conflicts. For such projects, there should never be a `using` directive in a header—this brings the corresponding names into any file that includes the header. This could result in name collisions and subtle, hard-to-find errors. Instead, use only fully qualified names in headers (for example, `std::cout` or `std::string`).

20.4.4 Aliases for `namespace` Names

`namespace`s can be **aliased**.[9] This might be useful when dealing with long namespace identifiers or nested namespaces. For example, assuming we have the `namespace` identifier `CPlusPlus20forProgrammers`, the statement

9. "Namespace aliases." Accessed March 11, 2022. https://en.cppreference.com/w/cpp/language/namespace_alias.

```
namespace CPP20FP = CPlusPlus20forProgrammers;
```

creates the shorter **namespace alias** `CPP20FP` for `CPlusPlus20forProgrammers`. Similarly, you could define an alias for the nested `namespace Inner` in Fig. 20.3 as follows:

```
namespace Innermost = Example::Inner;
```

17 20.4.5 C++17 Nested Namespaces

As of C++17, a nested namespace, such as `Inner` in lines 18–21, may be defined separately from its enclosing namespace. For example, we could remove lines 18–21 and define the nested namespace `Inner` as follows:

```
namespace Example::Inner {
   // define enumeration
   enum Years {fiscal1 = 2020, fiscal2, fiscal3};
}
```

The notation `Example::Inner` indicates that `Example` is the enclosing namespace and `Inner` is a nested namespace of `Example`.

20.5 Operator Keywords

The following table shows the **operator keywords** you can use in place of various C++ operators. You can use operator keywords if you have keyboards that do not support certain characters such as `!`, `&`, `^`, `~`, `|`, etc.

Operator	Operator keyword	Description
Logical operator keywords		
`&&`	`and`	logical AND
`||`	`or`	logical OR
`!`	`not`	logical NOT
Inequality operator keyword		
`!=`	`not_eq`	inequality
Bitwise operator keywords		
`&`	`bitand`	bitwise AND
`|`	`bitor`	bitwise inclusive OR
`^`	`xor`	bitwise exclusive OR

Operator	Operator keyword	Description
`~`	`compl`	bitwise complement
Bitwise assignment operator keywords		
`&=`	`and_eq`	bitwise AND assignment
`|=`	`or_eq`	bitwise inclusive OR assignment
`^=`	`xor_eq`	bitwise exclusive OR assignment

Figure 20.4 demonstrates several of the operator keywords. The program declares and initializes two `bool` variables (lines 7–8). Lines 13–19 use the logical operator keywords to perform various logical operations with `bool` variables `a` and `b`.

```
1   // fig20_04.cpp
2   // Demonstrating operator keywords.
3   #include <fmt/format.h>
4   #include <iostream>
5
6   int main() {
7      bool a{true};
8      bool b{false};
9
10     std::cout << fmt::format("a = {}; b = {}\n\n", a, b);
11
12     std::cout << "Logical operator keywords:\n"
13        << fmt::format("   a and a: {}\n", a and a)
14        << fmt::format("   a and b: {}\n", a and b)
15        << fmt::format("    a or a: {}\n", a or a)
16        << fmt::format("    a or b: {}\n", a or b)
17        << fmt::format("     not a: {}\n", not a)
18        << fmt::format("     not b: {}\n", not b)
19        << fmt::format("a not_eq b: {}\n", a not_eq b);
20  }
```

```
a = true; b = false

Logical operator keywords:
   a and a: true
   a and b: false
    a or a: true
    a or b: true
     not a: false
     not b: true
a not_eq b: true
```

Fig. 20.4 Demonstrating operator keywords.

20.6 Pointers to Class Members (`.*` and `->*`)

C++ provides the `.*` and `->*` operators for accessing class members via pointers. This capability is primarily for advanced C++ programmers. We provide only a mechanical example of using pointers to class members here. Figure 20.5 demonstrates the pointer-to-class-member operators. For answers to frequently asked questions about the .* and ->* operators, see https://isocpp.org/wiki/faq/pointers-to-members.

```
1  // fig20_05.cpp
2  // Demonstrating operators .* and ->*.
3  #include <iostream>
4
5  // class Test definition
6  class Test {
7  public:
8     Test(int x) : value{x} {};
9
10    void testFunction() {
11       std::cout << "testFunction called\n";
12    }
13
14    int value; // public data member
15 };
16
17 void arrowStar(Test* ptr); // prototype
18 void dotStar(Test* ptr); // prototype
19
20 int main() {
21    Test test{8};
22    arrowStar(&test); // pass address to arrowStar
23    dotStar(&test); // pass address to dotStar
24 }
25
26 // access member function of Test object using ->*
27 void arrowStar(Test* ptr) {
28    auto functionPtr{&Test::testFunction}; // pointer to a member function
29    (ptr->*functionPtr)(); // invoke function indirectly
30 }
31
32 // access members of Test object data member using .*
33 void dotStar(Test* ptr) {
34    auto valuePtr{&Test::value}; // pointer to a data member
35    std::cout << "value is " << (*ptr).*valuePtr << "\n"; // access value
36 }
```

```
testFunction called
value is 8
```

Fig. 20.5 Demonstrating operatprs `.*` and `->*`.

Class `Test`

SE The program declares class `Test` (lines 6–15) containing the `public` member function `testFunction` and the `public` data member `value`. **The client code can create pointers to class**

members for only those class members that are accessible to the client code. In this example, both member function `test` and data member `value` are publicly accessible.

`main` Function

Lines 17–18 provide prototypes for the functions `arrowStar` (defined in lines 27–30) and `dotStar` (defined in lines 33–36), which demonstrate the `->*` and `.*` operators, respectively. Line 21 creates a `Test` object named `test` and initializes its `value` member to `8`. Lines 22– 23 call functions `arrowStar` and `dotStar` with a pointer to the `test` object (i.e., `&test`).

`arrowStar` Function

Line 28 in function `arrowStar` declares and initializes variable `functionPtr` as a **pointer to a member function**. We declared the variable `auto` so the compiler can infer its type, which in this case is a pointer to a member function of class `Test` that takes no arguments and returns `void`. The type for a **pointer to a function** includes as part of its type both the function's **return type** and its **parameter list**. We initialize `functionPtr` with `&Test::testFunction`—that is, the address of class `Test`'s `testFunction` member. Note that the initializer uses the **address operator (&)** to get the member function's address. Line 29 invokes the member function stored in `functionPtr` using the `->*` operator. Because `functionPtr` is a pointer to a member of a class, the `->*` operator must be used rather than the `->` operator to invoke the function.

`dotStar` Function

Line 34 in `dotStar` declares and initializes `valuePtr`. Again, we declared the variable `auto` so the compiler can infer its type—in this case, a pointer to an `int` data member (`value`) of class `Test`. Line 35 dereferences the pointer `ptr` then uses the `.*` operator to access the member to which `valuePtr` points.

20.7 Case Study: Payroll System Using Polymorphism and Runtime Type Information — Downcasting, `dynamic_cast`, `typeid` and `type_info`

Section 10.9 presented a simple payroll system using runtime polymorphism. We defined an abstract base class `Employee` with concrete derived classes `SalariedEmployee` and `CommissionEmployee`. When processing `Employee` objects polymorphically in that example, we did not need to worry about the "specifics." We simply called each employee's `toString` and `earnings` member functions, and the correct functions were called based on the employee object's type.

CG For the current pay period, our fictitious company has decided to give a 10 percent base-salary increase to each `SalariedEmployee`. To adjust a `SalariedEmployee`'s base salary, we'll determine each `Employee`'s specific type at execution time, then act appropriately. Figure 20.6 uses **runtime type information (RTTI)** and **dynamic casting** to enable a program to determine an object's type at execution time. Before using this capability in your own programs, **be sure to read the C++ Core Guidelines' cautions about using RTTI at the end of this section**.

```
1  // fig20_06.cpp
2  // Demonstrating downcasting and runtime type information (RTTI).
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <typeinfo>
6  #include <vector>
```

```
 7   #include "Employee.h"
 8   #include "SalariedEmployee.h"
 9   #include "CommissionEmployee.h"
10
11   int main() {
12      // create derived-class objects
13      SalariedEmployee salaried{"John Smith", 800.0};
14      CommissionEmployee commission{"Sue Jones", 10000.0, .06};
15
16      // create and initialize vector of base-class pointers
17      std::vector<Employee*> employees{&salaried, &commission};
18
19      // polymorphically process each element in vector employees
20      for (Employee* employeePtr : employees) {
21         std::cout << fmt::format("{}\n", employeePtr->toString());
22
23         // determine whether employeePtr points to a SalariedEmployee;
24         // if not, dynamic_cast returns nullptr which evaluates to false
25         if (auto ptr{dynamic_cast<SalariedEmployee*>(employeePtr)}) {
26            double oldBaseSalary = ptr->getSalary();
27            std::cout << fmt::format("old salary: ${:.2f}\n", oldBaseSalary);
28            ptr->setSalary(1.10 * oldBaseSalary);
29            std::cout << fmt::format(
30               "new salary with 10% increase is: ${:.2f}\n",
31               ptr->getSalary());
32         }
33
34         std::cout << fmt::format("earned ${:.2f}\n\n",
35            employeePtr->earnings());
36      }
37
38      // display each object's type
39      for (const Employee* employeePtr : employees) {
40         std::cout << fmt::format("{}\n", typeid(*employeePtr).name());
41      }
42   }
```

```
name: John Smith
salary: $800.00
old salary: $800.00
new salary with 10% increase is: $880.00
earned $880.00

name: Sue Jones
gross sales: $10000.00
commission rate: 0.06
earned $600.00

16SalariedEmployee
18CommissionEmployee
```

Fig. 20.6 Demonstrating downcasting and runtime type information.

This program reuses classes `Employee`, `SalariedEmployee` and `CommissionEmployee` from Section 10.9. Lines 13–14 create one `SalariedEmployee` and one `CommissionEmployee` object. Line 17 uses the objects'

addresses to initialize the `vector` `employees`. Lines 20–36 iterate through `employees` and display each employee's information by polymorphically invoking member function `toString` (line 21).

Determining an Object's Type with `dynamic_cast`

As lines 20–36 encounter a `SalariedEmployee` object, we wish to increase its base salary by 10 percent. When processing `Employee`s polymorphically, we cannot know in advance whether `employeePtr` will point to a `SalariedEmployee` or a `CommissionEmployee` at any given time. So how do we identify `SalariedEmployee`s to increase their base salaries?

Err To accomplish this, we use operator **`dynamic_cast`** to determine whether the current `Employee`'s type is `SalariedEmployee`.[10] This operation is known as **downcasting**. The compiler will allow access to derived-class-only members from a base-class pointer aimed at a derived-class object **if we explicitly cast the base-class pointer down to a derived-class pointer**. Line 25 dynamically downcasts `employeePtr` from type `Employee*` to type `SalariedEmployee*`. **If `employeePtr` points to an object that *is a* `SalariedEmployee` object, then `ptr` is initialized with that object's address; otherwise, `ptr` is initialized with the value `nullptr`**. You also may use `dynamic_cast` with references. If the referenced object does not have the *is-a* relationship with the `dynamic_cast`'s specified type, the operator throws a **`bad_cast` exception**. Using `auto` in line 25 enables the compiler to infer `ptr`'s type (`SalariedEmployee*`) from the `dynamic_cast` expression.

10. "`dynamic_cast` conversion." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/language/dynamic_cast.

Err We use `dynamic_cast` here to check the underlying object's type. A `static_cast` would simply convert the `Employee*` to a `SalariedEmployee*`, ignoring the object's actual type. If we were to do that, the program would attempt to increase every `Employee`'s salary, resulting in undefined behavior for each object that's not a `SalariedEmployee`.

If the value returned by the `dynamic_cast` operator is not `nullptr`, the object is the correct type, and the `if` statement (lines 25–32) performs the special processing required for the `SalariedEmployee` object. Lines 26, 28 and 31 invoke `SalariedEmployee` functions `getSalary` and `setSalary` to retrieve and update the employee's base salary.

Calculating the Current `Employee`'s Earnings

Lines 34–35 invoke `earnings` on the current employee being processed. Recall that `earnings` is declared `virtual` in the base class, so the program polymorphically invokes each employee's `earnings` member function.

Displaying an `Employee`'s Type

Lines 39–41 display each employee's object type. Operator **`typeid`**[11] (line 40)—which requires the header **`<typeinfo>`** (line 5)—returns a reference to a **`type_info`**[12] object containing information about its operand's type, including the type's name. The `type_info` member function **`name`** returns an implementation-depending pointer-based string containing the `typeid` argument's type name. The sample output was produced with `g++`, which precedes each class's name with the number of characters in the class name (e.g., `"16SalariedEmployee"`)—`clang++` produces the same output. Visual C++ produces:

11. "`typeid` operator." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/language/typeid.
12. "`type_info` operator." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/types/type_info.

```
class SalariedEmployee
class CommissionEmployee
```

Compilation Errors That We Avoided By Using `dynamic_cast`

We avoid several compilation errors in this example by downcasting the `Employee` pointer to a `SalariedEmployee` pointer (line 25). If lines 26, 28 and 31 used a base-class pointer from the current element of `employees` rather than a derived-class pointer to invoke derived-class-only functions `getSalary` and `setSalary`, we'd receive a compilation error at each of these lines. As you learned in Section 10.6.3, attempting to invoke derived-class-only functions through a base-class pointer is not allowed.

RTTI Cautions

The C++ Core Guidelines indicate that `dynamic_cast` is overused and should be used only "where class hierarchy navigation is unavoidable."[13] For more information on the primary uses of RTTI and its alternatives, see core guidelines C.146–C.148:

- C.146—**"Use `dynamic_cast` where class hierarchy navigation is unavoidable"**: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rhdynamic_cast
- C.147—**"Use `dynamic_cast` to a reference type when failure to find the required class is considered an error"**: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-ref-cast
- C.148—**"Use `dynamic_cast` to a pointer type when failure to find the required class is considered a valid alternative"**: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-ptr-cast

13. C++ Core Guidelines, "C.146: Use `dynamic_cast` where class hierarchy navigation is unavoidable." Accessed March 12, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rh-dynamic_cast.

11 20.8 Inheriting Base-Class Constructors

Base-class constructors, destructors and overloaded assignment operators are not inherited by derived classes. However, derived-class constructors, destructors and overloaded assignment operators can call their base-class versions.

Sometimes a derived class's constructors simply pass the constructor arguments to a base-class constructor with the same parameters. In such cases, you can specify that a derived class should inherit its base class's constructors. To do so, include a `using` declaration of the following form anywhere in the derived-class definition:

```
using BaseClass::BaseClass;
```

We suggest placing this where you'd typically place the constructor's prototype. In the preceding declaration, *BaseClass* is the base class's name. The compiler generates a corresponding derived-class constructor for each base-class constructor that calls that base-class one. The generated constructors have the derived class's name and perform only default initialization for the derived class's additional data members, if any.

When you inherit constructors:[14,15]

14. "Using-declaration." Accessed March 10, 2022. https://en.cppreference.com/w/cpp/language/using_declaration
15. C++ Standard, "9.9 The `using` declaration." Accessed March 10, 2022. https://timsongcpp.github.io/cppwp/n4861/namespace.udecl.

- Each generated constructor has the same access specifier (`public`, `protected` or `private`) as its corresponding base-class constructor.
- If a constructor is deleted in the base class by placing `= delete` in its prototype, the corresponding derived-class constructor also is deleted.
- If the derived class does not explicitly define constructors, the compiler still generates a default constructor in the derived class.
- A given base-class constructor is not accessible if a constructor that you explicitly define in the derived class has the same parameter list.
- A base-class constructor's default arguments are not inherited. Instead, the compiler generates overloaded constructors in the derived class. For example, if the base class declares the constructor

```
BaseClass(int x = 0, double y = 0.0);
```

the compiler generates the following derived-class constructors without default arguments

```
DerivedClass();

DerivedClass(int x);

DerivedClass(int x, double y);
```

These each call the *BaseClass* constructor that specifies the default arguments.

17 20.9 C++17 and C++20: `[[nodiscard]]` Attribute

20 **Some functions return values that you should not ignore.** For example, Section 2.7 introduced the `string` member function `empty`. When you want to know whether a `string` is empty, you must not only call `empty` but also check its return value in a condition, such as:

```
if (s.empty()) {
   // do something because the string s is empty
}
```

20 SE In C++20, `string`'s `empty` function is declared with the **`[[nodiscard]]` attribute,**[16] **so the compiler issues a warning if the caller does not use the return value**. Since C++17, many library functions have been enhanced with `[[nodiscard]]` to help you write correct code.

16. Section 9.12.8 of the "Working Draft, Standard for Programming Language C++." Accessed May 10, 2020. http://wg21.link/n4861.

20 C++20's `[[nodiscard("with reason")]]` Attribute

One problem with C++17's `[[nodiscard]]` attribute is that it did not specify why you should not ignore a given function's return value. So, in C++20, you can include a descriptive message that the compiler will display as part of the warning message, as in:

```
[[nodiscard("Insight goes here")]]
```

20 You may also use this attribute on your own function definitions. Figure 20.7 shows a `cube` function declared with C++20's `[[nodiscard]]` version. You place the attribute before the return type—typically on a line by itself for readability (line 4). Line 10 calls `cube` but does not use the returned value. The three outputs show the compiler warnings from Visual C++, `clang++` and `g++`, respectively. These are just warnings, so the program still compiles and runs. The primary purpose of this example is to demonstrate the compiler warning messages produced by `[[nodiscard]]`. This program does not produce any output when it runs because it ignores function `cube`'s return value when it should not.

```
1   // fig20_07.cpp
2   // [[nodiscard]] attribute.
3
4   [[nodiscard("Do not ignore! Otherwise, you won't know the cube of x")]]
5   int cube(int x) {
6      return x * x * x;
7   }
8
9   int main() {
10     cube(10); // generates a compiler warning
11  }
```

```
C:\Users\pauldeitel\examples\ch20\fig20_07.cpp(10,8): warning C4858: discard-
ing return value: Do not ignore! Otherwise, you won't know the cube of x.
fig20_07.cpp:10:4: warning: ignoring return value of function declared with
'nodiscard' attribute: Do not ignore! Otherwise, you won't know the cube of
x. [-Wunused-result]
   cube(10); // generates a compiler warning
   ^~~~ ~~
1 warning generated.
fig20_07.cpp: In function 'int main()':
fig20_07.cpp:10:8: warning: ignoring return value of 'int cube(int)', de-
clared with attribute 'nodiscard': 'Do not ignore! Otherwise, you won't know
the cube of x.' [-Wunused-result]
   10 |   cube(10); // generates a compiler warning
      |   ~~~~^~~~
fig20_07.cpp:5:5: note: declared here
    5 | int cube(int x) {
      |     ^~~~
```

Fig. 20.7 `[[nodiscard]]` attribute.

20.10 `shared_ptr` and `weak_ptr` Smart Pointers

SE Err 11 Sections 11.4–11.5 introduced dynamic memory allocation and smart pointers. You learned that many common bugs in C and C++ code are related to pointers and dynamically allocated memory and that smart pointers help you avoid such errors by providing additional functionality that strengthens memory allocation and deallocation. Smart pointers also help you write **exception-safe code** (introduced in Section 12.3). If a program throws an exception before `delete` has been called on a pointer to dynamically allocated memory, a memory leak occurs. On the other hand, **smart pointers use resource acquisition is initialization (RAII**; Section 11.5). A smart pointer is an object, so **when a smart pointer goes out of scope for any reason, including an exception, the smart pointer's destructor will still be called, helping avoid memory leaks**. In addition

to class template `unique_ptr` (discussed in Section 11.5), C++11 introduced other smart pointer options with additional functionality.

20.10.1 Reference Counted `shared_ptr`

11 C++11 **`shared_ptr`s** (header `<memory>`) hold an internal pointer to a resource (e.g., a dynamically allocated object) that may be shared throughout a program.[17] You can have many `shared_ptr`s to the same resource. As the type name implies, **`shared_ptr`s share a resource**—if you change the resource with one `shared_ptr`, the changes also will be "seen" by the other `shared_ptr`s. The internal pointer is deleted when the last `shared_ptr` to the resource is destroyed. Internally, `shared_ptr`s use **reference counting** to determine how many `shared_ptr`s point to the resource. Each time you create a new `shared_ptr` to a resource, the **reference count** increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted and the memory is released.

17. "`std::shared_ptr`." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/memory/shared_ptr.

`shared_ptr`s vs. `unique_ptr`s

CG The C++ Core Guidelines indicate that you should prefer a `unique_ptr` over a `shared_ptr` if there will be only one owner of a given resource at a time.[18]

18. C++ Core Guidelines, "F.27: Use a `shared_ptr<T>` to share ownership." Accessed March 18, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rf-shared_ptr.

Customizing How Shared Resources Are Destroyed

`shared_ptr`s also allow you to determine how the resource will be destroyed. For most dynamically allocated objects, `delete` is used. However, some resources require more complex cleanup. In that case, you can supply a custom **deleter** function (a function pointer, lambda or function object) to the `shared_ptr` constructor. The deleter specifies how to destroy the resource. When the reference count reaches zero, and the resource is ready to be destroyed, the `shared_ptr` calls the custom deleter function.

Example Using `shared_ptr`

Fig. 20.8 defines a simple `Book` class (lines 12–17) with a `string` representing the `Book`'s title. `Book`'s destructor (line 15) displays a message indicating which `Book` object is being destroyed. We use this class to demonstrate the basic functionality of `shared_ptr`.

```
1  // fig20_08.cpp
2  // Demonstrate shared_ptrs.
3  #include <algorithm>
4  #include <fmt/format.h>
5  #include <iostream>
6  #include <memory>
7  #include <string>
8  #include <string_view>
9  #include <vector>
10
11 // class Book
```

```
12  class Book {
13  public:
14     explicit Book(std::string_view bookTitle) : title{bookTitle} {}
15     ~Book() {std::cout << fmt::format("Destroying Book: {}\n", title);}
16     std::string title; // title of the Book
17  };
18
19  // a custom delete function for a pointer to a Book
20  void deleteBook(Book* book) {
21     std::cout << "Custom deleter for a Book, ";
22     delete book; // delete the Book pointer
23  }
24
25  // compare the titles of two Books for sorting
26  bool compareTitles(
27     std::shared_ptr<Book> ptr1, std::shared_ptr<Book> ptr2) {
28     return (ptr1->title < ptr2->title);
29  }
30
31  int main() {
32     // create a shared_ptr to a Book and display the reference count
33     std::shared_ptr<Book> bookPtr{
34        std::make_shared<Book>("C++ How to Program")};
35     std::cout << fmt::format("Reference count for Book {} is: {}\n",
36        bookPtr->title, bookPtr.use_count());
37
38     // create another shared_ptr to the Book and display reference count
39     std::shared_ptr<Book> bookPtr2{bookPtr};
40     std::cout << fmt::format("Reference count for Book {} is: {}\n",
41        bookPtr->title, bookPtr.use_count());
42
43     // change the Book's title and access it from both pointers
44     bookPtr2->title = "Java How to Program";
45     std::cout << fmt::format(
46        "Updated Book title:\nbookPtr: {}\nbookPtr2: {}\n",
47        bookPtr->title, bookPtr2->title);
48
49     // create a std::vector of shared_ptrs to Books (BookPtrs)
50     std::vector<std::shared_ptr<Book>> books{
51        std::make_shared<Book>("C How to Program"),
52        std::make_shared<Book>("Intro to Python"),
53        std::make_shared<Book>("C# How to Program"),
54        std::make_shared<Book>("C++ How to Program")};
55
56     // print the Books in the vector
57     std::cout << "\nBooks before sorting:\n";
58     for (auto book : books) {
59        std::cout << book->title << "\n";
60     }
61
62     // sort the vector by Book title and print the sorted vector
63     std::sort(books.begin(), books.end(), compareTitles);
64     std::cout << "\nBooks after sorting:\n";
65     for (auto book : books) {
66        std::cout << book->title << "\n";
```

```
67      }
68
69      // create a shared_ptr with a custom deleter
70      std::cout << "\nshared_ptr with a custom deleter.\n";
71      std::shared_ptr<Book> bookPtr3{
72         new Book("Android How to Program"), deleteBook};
73      bookPtr3.reset(); // release the Book this shared_ptr manages
74
75      // shared_ptrs are going out of scope
76      std::cout << "\nEnd of main: shared_ptr objects going out of scope.\n";
77   }
```

```
Reference count for Book C++ How to Program is: 1
Reference count for Book C++ How to Program is: 2
Updated Book title:
bookPtr: Java How to Program
bookPtr2: Java How to Program

Books before sorting:
C How to Program
Intro to Python
C# How to Program
C++ How to Program

Books after sorting:
C How to Program
C# How to Program
C++ How to Program
Intro to Python

shared_ptr with a custom deleter.
Custom deleter for a Book, Destroying Book: Android How to Program

End of main: shared_ptr objects going out of scope.
Destroying Book: C How to Program
Destroying Book: C# How to Program
Destroying Book: C++ How to Program
Destroying Book: Intro to Python
Destroying Book: Java How to Program
```

Fig. 20.8 `shared_ptr` example program.

Creating `shared_ptr`s

11 CG The program uses C++11 `shared_ptr`s to manage several `Book` instances. Lines 33–34 use standard library function `make_shared` to create a `shared_ptr` to a dynamically allocated `Book` object with the title `"C++ How to Program"`.[19] This is the recommended way to create a `shared_ptr`.[20] You can further simplify this statement by declaring `bookPtr` with `auto`. The `shared_ptr` manages the `Book` object and initially sets its reference count to 1—that is, one `shared_ptr` currently refers to the dynamically allocated object. Class `shared_ptr` provides various constructors—if a new `shared_ptr` is initialized with an existing one, they both share ownership of the resource and the reference count increases by 1. If you make multiple `shared_ptr`s with the same pointer, the `shared_ptr`s won't acknowledge each other, and the reference count will be wrong. When the `shared_ptr`s are destroyed, they decrease the reference count by one. The `shared_ptr` destructor call that decrements the shared resource's reference count to 0 also deletes the resource.

19. "std::make_shared, std::make_shared_for_overwrite." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared.
20. C++ Core Guidelines, "R.22: Use make_shared() to make shared_ptrs." Accessed March 12, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-make_shared.

Manipulating shared_ptrs

Lines 35–36 display the Book's title and the number of shared_ptrs referencing that object—that is, the current reference count. We use the -> operator to access the Book's data member title—shared_ptrs overload the pointer operators -> and * so you can use them like raw pointers. The shared_ptr member function **use_count** returns the reference count.

Line 39 creates another shared_ptr to the same Book using the shared_ptr constructor that receives a shared_ptr argument—this increments the Book's reference count by one. You also can use shared_ptr's assignment operator (=) to create a shared_ptr to the same resource. Lines 40–41 display the Book's title and reference count again to show that the reference count increased by one when we created the second shared_ptr.

As mentioned earlier, changes made to the resource of a shared_ptr are "seen" by all shared_ptrs to that resource. Line 44 uses bookPtr2 to change the Book's title, then lines 45–47 display the Book's title using both bookPtr and bookPtr2 to show that bookPtr "sees" the change.

Manipulating shared_ptrs in an STL Container

Next, we demonstrate using shared_ptrs in an STL container. Lines 50–54 create a vector of shared_ptr<Book> objects and add four elements, using make_shared to dynamically allocate each Book and return a shared_ptr to it. Lines 58–60 display the vector's initial contents. Then, line 63 sorts the Books by title, using the function compareTitles (lines 26–29) to customize how the sort algorithm compares Book objects—in this case, by comparing the Books' titles. Lines 65–67 display the vector's sorted contents.

shared_ptr Custom Deleter

Lines 71–72 create a shared_ptr with a custom deleter function deleteBook (lines 20–23), which is passed to the shared_ptr constructor as the second argument. A custom deleter function must take one argument of the shared_ptr's internal pointer type. When the last shared_ptr for a given Book object is destroyed, the shared_ptr passes its internal Book* to the custom deleter, which is responsible for deleting the managed resource. In this example, deleteBook displays a message showing that the custom deleter was called, then deletes the dynamically allocated Book object.

One use for custom deleters is when using third-party C libraries. Rather than providing a class with a constructor and destructor as a C++ library would, C libraries frequently provide one function that returns a pointer to a struct representing a resource and another that does the necessary cleanup when the resource is no longer needed. Using a custom deleter allows you to use a shared_ptr to keep track of the resource and still ensure that it's freed correctly.

Resetting a shared_ptr

Line 73 calls the shared_ptr member function **reset**, which sets the shared_ptr to nullptr, so it no longer points to a shared resource. Since bookPtr3 was the only shared_ptr to the Book object created in lines 71–72, the shared_ptr calls its custom deleter to release the resource. Other versions of reset allow you to pass a new resource to manage.

`shared_ptr`s Are Destroyed When They Go Out of Scope

When `main` terminates, all the `shared_ptr`s and the `vector` in this example go out of scope and are destroyed. Before the `vector` is destroyed, its `shared_ptr` elements are destroyed. The output shows that each `Book` object is destroyed automatically by the `shared_ptr`s.

11 20.10.2 `weak_ptr`: `shared_ptr` Observer

Err A `weak_ptr` points to the resource managed by a `shared_ptr` without assuming any responsibility for it.[21] A `shared_ptr`'s reference count does not increase when a `weak_ptr` references the shared resource. Thus, **a `shared_ptr`'s resource can be deleted even if there are still `weak_ptr`s pointing to it. When the last `shared_ptr` to a resource is destroyed, any remaining `weak_ptr`s are set to `nullptr`.** The C++ Core Guidelines say to use `weak_ptr`s "to break cycles of `shared_ptr`s."[22] As we'll demonstrate later in this section, this helps avoid memory leaks caused by circular references.

21. "`std::weak_ptr`." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/memory/weak_ptr.

22. C++ Core Guidelines, "R.24: Use `std::weak_ptr` to break cycles of `shared_ptr`s." Accessed March 12, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Rr-weak_ptr.

A `weak_ptr` cannot directly access the resource it points to—you must create a `shared_ptr` from the `weak_ptr` to access the resource. This enables a program to determine whether the resource still exists. There are two ways to create a `shared_ptr` from a `weak_ptr`:

- Err You can pass the `weak_ptr` to the `shared_ptr` constructor, which creates a `shared_ptr` to the resource being pointed to by the `weak_ptr` and properly increases the reference count. If the resource has already been deleted, the `shared_ptr` constructor will throw a `bad_weak_ptr` exception.
- You can also call the `weak_ptr` member function `lock`, which returns a `shared_ptr` to the `weak_ptr`'s resource.[23] If the `weak_ptr` does not currently point to a resource, `lock` returns an empty `shared_ptr` (i.e., a `shared_ptr` with its internal pointer set to `nullptr`). You should use `lock` if an empty `shared_ptr` isn't considered an error in the context of your application.

23. "`std::weak_ptr::lock`." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/memory/weak_ptr/lock.

Err One typical use-case of `weak_ptr`s is in **circularly referential data**—a situation in which two objects refer to each other internally, as the example in Figs. 20.9–20.12 demonstrates. Another use-case is avoiding **dangling pointers**[24]—that is, pointers to dynamically allocated objects that are subsequently deleted. Using such a pointer is undefined behavior that often crashes a program. **Because a `weak_ptr` cannot directly access the resource it points to, the resource can be deleted.** If the program subsequently attempts to use the `weak_ptr` 's `lock` function to get a `shared_ptr`, `lock` returns `nullptr` to indicate that the resource no longer exists.

24. "When is `std::weak_ptr` useful?" Accessed March 17, 2022. https://stackoverflow.com/questions/12030650/when-is-stdweak-ptr-useful.

Class Definitions for Classes `Author` and `Book`

We'll demonstrate `weak_ptr`s using classes `Author` and `Book` (Figs. 20.9–20.12). Each class contains pointers to an object of the other class. We also use `public` data members here only to simplify the code. This creates a **circular type reference** between the class definitions:

- class `Author` defines a `weak_ptr` and a `shared_ptr` to a `Book` (lines 20–21 in Fig. 20.9), and
- class `Book` defines a `weak_ptr` and a `shared_ptr` to an `Author` (lines 20–21 in Fig. 20.10).

To declare these pointers, each class needs to know that the other exists, which you usually accomplish by including the corresponding class's header. However, we cannot do that here because class `Author` depends on class `Book`, which, in turn, depends on class `Author`.
Including `Book.h` in `Author.h` and including `Author.h` in `Book.h` results in compilation errors.
Interestingly, declaring a pointer does not require a complete type definition—it simply requires knowing that the type exists. This is the purpose of the **forward class declarations** in line 8 of Figs. 20.9 and 20.10. Once you need to use a pointer to interact with an object, then you need the full class definition, as you'll see in the `.cpp` files for classes `Author` (Fig. 20.11) and `Book` (Fig. 20.12).

```
1  // Fig. 20.9: Author.h
2  // Author class definition.
3  #pragma once
4  #include <memory>
5  #include <string>
6  #include <string_view>
7
8  class Book; // forward declaration of class Book
9
10 // Author class definition
11 class Author {
12 public:
13    explicit Author(std::string_view authorName);
14    ~Author();
15
16    // print the title of the Book this Author wrote
17    void printBookTitle();
18
19    std::string name; // name of the Author
20    std::weak_ptr<Book> weakBookPtr; // Book the Author wrote
21    std::shared_ptr<Book> sharedBookPtr; // Book the Author wrote
22 };
```

Fig. 20.9 `Author` class definition.

```
1  // Fig. 20.10: Book.h
2  // Book class definition.
3  #pragma once
4  #include <memory>
5  #include <string>
6  #include <string_view>
7
8  class Author; // forward declaration of class Author
9
10 // Book class definition
11 class Book {
12 public:
13    explicit Book(std::string_view bookTitle);
14    ~Book();
15
16    // print the name of this Book's Author
17    void printAuthorName();
18
19    std::string title; // title of the Book
20    std::weak_ptr<Author> weakAuthorPtr; // Author of the Book
```

```
21      std::shared_ptr<Author> sharedAuthorPtr; // Author of the Book
22   };
```

Fig. 20.10 `Book` class definition.

Err Using classes `Author` and `Book`, we'll show that setting an `Author` object's `shared_ptr` to point to a `Book` object and setting that `Book` object's `shared_ptr` to point back to the `Author` object **creates a memory leak**. Then, we'll show how we can use the `weak_ptr`s to fix this problem.

Member Function Definitions for Classes `Author` and `Book`

Figures 20.11 and 20.12 define `Author`'s and `Book`'s member functions, respectively. For an `Author` object's member functions to manipulate a corresponding `Book` object through a pointer, class `Author` must know class `Book`'s complete definition. So line 9 in Fig. 20.11 includes `Book.h`. Similarly, for a `Book` object's member functions to manipulate a corresponding `Author` object through a pointer, class `Book` must know class `Author`'s complete definition. So line 8 in Fig. 20.12 includes `Author.h`.

```
1   // Fig. 20.11: Author.cpp
2   // Author member-function definitions.
3   #include <fmt/format.h>
4   #include <iostream>
5   #include <memory>
6   #include <string>
7   #include <string_view>
8   #include "Author.h"
9   #include "Book.h"
10
11  Author::Author(std::string_view authorName) : name(authorName) {}
12
13  Author::~Author() {
14     std::cout << fmt::format("Destroying Author: {}\n", name);
15  }
16
17  // print the title of the Book this Author wrote
18  void Author::printBookTitle() {
19     // if weakBookPtr.lock() returns a non-empty shared_ptr
20     if (std::shared_ptr<Book> bookPtr{weakBookPtr.lock()}) {
21        // show the reference count increase and print the Book's title
22        std::cout << fmt::format("Reference count for Book {} is {}\n",
23           bookPtr->title, bookPtr.use_count());
24        std::cout << fmt::format("Author {} wrote the book {}\n",
25           name, bookPtr->title);
26     }
27     else { // weakBookPtr points to NULL
28        std::cout << "This Author has no Books.\n";
29     }
30  }
```

Fig. 20.11 Author member-function definitions.

```
1   // Fig. 20.12: Book.cpp
2   // Book member-function definitions.
3   #include <fmt/format.h>
4   #include <iostream>
```

```
 5   #include <memory>
 6   #include <string>
 7   #include <string_view>
 8   #include "Author.h"
 9   #include "Book.h"
10
11   Book::Book(std::string_view bookTitle) : title(bookTitle) {}
12
13   Book::~Book() {
14      std::cout << fmt::format("Destroying Book: {}\n", title);
15   }
16
17   // print the name of this Book's Author
18   void Book::printAuthorName() {
19      // if weakAuthorPtr.lock() returns a non-empty shared_ptr
20      if (std::shared_ptr<Author> authorPtr{weakAuthorPtr.lock()}) {
21         // show the reference count increase and print the Author's name
22         std::cout << fmt::format("Reference count for Author {} is {}\n",
23            authorPtr->name, authorPtr.use_count());
24         std::cout << fmt::format("The book {} was written by {}\n",
25            title, authorPtr->name);
26      }
27      else { // weakAuthorPtr points to NULL
28         std::cout << "This Book has no Author.\n";
29      }
30   }
```

Fig. 20.12 Book member-function definitions.

Both classes define destructors that display a message to indicate when an object of either class is destroyed (Figs. 20.11 and 20.12, lines 13–15). Class `Author`'s `printBook-Title` function (Fig. 20.11, lines 18–30) displays its `Book`'s `title` and reference count (lines 22–23), then shows that the `Author` wrote that `Book` (lines 24–25). Similarly, class `Book`'s `printAuthorName` function (Fig. 20.12, lines 18–30) displays its `Author`'s `name` and reference count (lines 22–23), then shows that the `Book` was written by that `Author` (lines 24–25).

Recall that you can't access a resource directly through a `weak_ptr`. For `printBook-Title` and `printAuthorName` to perform the tasks in lines 22–25 of each, these functions must first create a `shared_ptr` from the `weak_ptr` data member by calling the `weak_ptr`'s `lock` function (line 20 in each figure). If the `weak_ptr` does not reference a resource, the `lock` function returns a `shared_ptr` containing `nullptr`, which evaluates to `false` in line 20. Otherwise, the new `shared_ptr` contains a valid pointer to the `weak_ptr`'s resource, the condition in line 20 evaluates to `true`, and lines 22–25 of each figure can access the corresponding resource. Lines 22–23 of each function display the `shared_ptr`'s reference count to show that creating the new `shared_ptr` **increased the reference count for the corresponding `Book`** (Fig. 20.11) **or `Author`** (Fig. 20.12). The new `shared_ptr` created in line 20 of each figure is a local object, so it's destroyed when its enclosing block ends. At that point, the corresponding reference count decreases by one.

`main` Function

Err Figure 20.13 demonstrates the memory leak caused by the circular reference between `Author` and `Book`. Lines 11–14 create `shared_ptr`s to an object of each class. Line 17 sets the `Books`'s `weak_ptr` data member to point to the `Author`, and line 18 sets the `Author`'s `weak_ptr` data

member to point to the `Book`. Similarly, lines 21–22 set the `shared_ptr` data members of each object. The `Author` and `Book` objects now reference each other.

```
1  // fig20_13.cpp
2  // Demonstrate use of weak_ptr.
3  #include <fmt/format.h>
4  #include <iostream>
5  #include <memory>
6  #include "Author.h"
7  #include "Book.h"
8
9  int main() {
10    // create a Book and an Author
11    std::shared_ptr<Book> bookPtr(
12       std::make_shared<Book>("C++ How to Program"));
13    std::shared_ptr<Author> authorPtr(
14       std::make_shared<Author>("Deitel & Deitel"));
15
16    // reference the Book and Author to each other
17    bookPtr->weakAuthorPtr = authorPtr;
18    authorPtr->weakBookPtr = bookPtr;
19
20    // set the shared_ptr data members to create the memory leak
21    bookPtr->sharedAuthorPtr = authorPtr;
22    authorPtr->sharedBookPtr = bookPtr;
23
24    // reference count for bookPtr and authorPtr is two
25    std::cout << fmt::format("Reference count for Book {} is {}\n",
26       bookPtr->title, bookPtr.use_count());
27    std::cout << fmt::format("Reference count for Author {} is {}\n\n",
28       authorPtr->name, authorPtr.use_count());
29
30    // access the cross references to print the data they point to
31    std::cout << "Access Author name and Book title via weak_ptrs:\n";
32    bookPtr->printAuthorName();
33    std::cout << "\n";
34    authorPtr->printBookTitle();
35
36    // reference count for each shared_ptr is two
37    std::cout << fmt::format("Reference count for Book {} is {}\n",
38       bookPtr->title, bookPtr.use_count());
39    std::cout << fmt::format("Reference count for Author {} is {}\n\n",
40       authorPtr->name, authorPtr.use_count());
41
42    // the shared_ptrs go out of scope, the Book and Author are destroyed
43    std::cout << "End of main. shared_ptrs going out of scope.\n";
44 }
```

```
Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

Access Author name and Book title via weak_ptrs:
Reference count for Author Deitel & Deitel is 3
The book C++ How to Program was written by Deitel & Deitel
```

```
Reference count for Book C++ How to Program is 3
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

End of main. shared_ptrs going out of scope.
```

Fig. 20.13 `shared_ptr`s cause a memory leak in circularly referential data.

Next, lines 25–28 display the reference counts for the `shared_ptr`s to show that each object is referenced by two `shared_ptr`s:

- the `Book` object is referenced by the `shared_ptr` in lines 11–12 and by the one in the `Author` object (line 22), and
- the `Author` object is referenced by the `shared_ptr` in lines 13–14 and by the one in the `Book` object (line 21).

Note that the `weak_ptr`s don't affect the reference counts. Line 32 calls the `Book` object's `printAuthorName` function to display the information stored in the `Book`'s `weak_ptr` data member. Line 34 calls the `Author` object's `printBookTitle` function to display the information stored in the `Author`'s `weak_ptr` data member. Each of these functions also displays the fact that another `shared_ptr` was created during the function call increasing the `Book`'s and `Author`'s reference counts to `3`. Finally, lines 37–40 display the `Book`'s and `Author`'s reference counts again to show that the additional `shared_ptr`s created in the `printAuthorName` and `printBookTitle` were destroyed when the functions finished executing, decreasing each object's reference count to `2`. Then, `main` terminates.

Memory Leak

Err At the end of `main`, the `shared_ptr`s to the instances of `Author` and `Book` we created go out of scope and are destroyed, but **notice that the output doesn't show the destructors for classes `Author` and `Book`**. This program has a **memory leak**—the `Author` and `Book` objects aren't destroyed because of their `shared_ptr` data members. When `bookPtr` is destroyed at the end of the `main` function, the reference count for the object of class `Book` becomes one—the object of `Author` still has a `shared_ptr` to the `Book` object, so it isn't deleted. When `authorPtr` goes out of scope and is destroyed, the reference count for the object of class `Author` also becomes one—the `Book` object still has a `shared_ptr` to the `Author` object. Neither object is deleted because the reference count for each is still one.

Fixing the Memory Leak

Now, comment out lines 21–22 in `main` by placing `//` at the beginning of each line. This prevents the code from setting the `shared_ptr` data members for classes `Author` and `Book`. Recompile the code and run the program again to produce the following output. We bolded the last two lines showing the `Author` and `Book` objects were properly destroyed.

```
Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

Access Author name and Book title via weak_ptrs:
Reference count for Author Deitel & Deitel is 2
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 2
```

```
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

End of main. shared_ptrs going out of scope.
Destroying Author: Deitel & Deitel
Destroying Book: C++ How to Program
```

Notice that the initial reference count for each object is now `1` instead of `2` because we don't set the `shared_ptr` data members. The last two lines of the output show that the `Author` and `Book` objects were destroyed at the end of `main`. We eliminated the memory leak by using only the `weak_ptr` data members, which don't affect the reference count but still allow us to access the resource when we need it by creating a temporary `shared_ptr` to the resource. When the `shared_ptr`s we created in `main` are destroyed, **their reference counts become `0`**, and the instances of classes `Author` and `Book` are deleted properly.

11 20.11 Trailing Return Types for Functions

Section 14.3 introduced lambda expressions. As you saw, lambdas that return values are declared with trailing return types in which the return type is specified to the right of the parameter list. Trailing-return-type syntax also can be used with functions and member functions. To specify a trailing return type, place the keyword `auto` before the function name, then follow the function's parameter list with `->` and the return type. For example, to specify a trailing return type for a function template `maximum` that finds the largest of three values of the same type, `T`, you'd write

```
template <typename T>
auto maximum(T x, T y, T z) -> T {
   // statements
}
```

As you build more complex function templates, there are cases for which only trailing return types are allowed. Such complex function templates are beyond this book's scope. For more information on trailing return types, see the C++ Standard Section 9.3.3.5, "Functions" at

```
https://timsong-cpp.github.io/cppwp/n4861/dcl.fct
```

20.12 `decltype`

11 C++11's **`decltype`** operator enables the compiler to determine an expression's type at compile time.[25] If the expression is a function call, `decltype` determines the function's return type, which for a function template often changes based on the type(s) used to specialize the template.

25. "`decltype` specifier." Accessed March 12, 2022. https://en.cppreference.com/w/cpp/language/decltype.

The `decltype` operator is particularly useful when working with complex template types for which it's often difficult to provide, or even determine, the proper type declaration. Rather than writing a complex type declaration, for example, representing the return type of a function, you can place in the parentheses of `decltype` an expression that returns the complex type and let the compiler "figure it

out.” Determining types at compile time with `decltype` is used frequently in C++ standard library class templates and in template metaprogramming.

The format of a `decltype` expression is

```
decltype(expression)
```

14 The *expression* is not evaluated. `decltype` is commonly used with trailing return types in function definitions that return complex types. C++14 added `decltype(auto)` to infer a function’s return type as in:

```
decltype(auto) functionName(parameters)
```

For more information on `decltype`, see the C++ Standard Section 9.2.8.4, “Decltype specifiers” at

```
https://timsong-cpp.github.io/cppwp/n4861/dcl.type.decltype
```

20.13 `initializer_list` Class Template

Previously, you learned how to use C++11’s braced initialization capabilities to initialize variables and containers. You can also define functions and constructors that receive braced initializers as arguments. To do so, you specify an `initializer_list` parameter. Figure 20.14 defines a function template `sum` (lines 9–19) that receives an `initializer_list` (line 10) and sums its elements. An initializer list can be used with the range-based `for` statement (lines 14–16) to iterate through all items in the `initializer_list`. In `main`, lines 24, 28 and 34 demonstrate passing a braced initializer to `sum`’s `initializer_list` parameter.

```
1  // fig20_14.cpp
2  // Summing the elements of a braced initializer
3  #include <fmt/format.h>
4  #include <initializer_list>
5  #include <iostream>
6  #include <string>
7
8  // sum the elements of an initializer_list
9  template <typename T>
10 T sum(std::initializer_list<T> list) {
11    T total{}; // value initialize total based on type T
12
13    // sum all the elements in list; requires += operator for type T
14    for (auto item : list) {
15       total += item;
16    }
17
18    return total;
19 }
20
21 int main() {
22    // display the sum of four ints contained in a braced initializer
```

```
23      std::cout << fmt::format("The sum of {} is: {} \n",
24         "{1, 2, 3, 4}", sum({1, 2, 3, 4}));
25
26      // display the sum of three doubles contained in a braced initializer
27      std::cout << fmt::format("The sum of {} is: {}\n",
28         "{1.1, 2.2, 3.3}", sum({1.1, 2.2, 3.3}));
29
30      // display the sum of two strings contained in a braced initializer
31      std::string s1{"Happy "};
32      std::string s2{"birthday!"};
33      std::cout << fmt::format("The sum of {} is: {}\n",
34         "{\"Happy \", \"birthday!\"}", sum({s1, s2}));
35   }
```

```
The sum of {1, 2, 3, 4} is: 10
The sum of {1.1, 2.2, 3.3} is: 6.6
The sum of {"Happy ", "birthday!"} is: Happy birthday!
```

Fig. 20.14 Summing the elements of a braced initializer.

20 20.14 C++20: `[[likely]]` and `[[unlikely]]` Attributes

Perf Today's compilers use sophisticated optimization techniques[26] to tune your code's performance. C++20 introduces the attributes `[[likely]]` and `[[unlikely]]` that enable you to provide additional hints to help compilers optimize `if`, `if`...`else` and `switch` statements code for better performance.[27] These attributes indicate paths of execution that are likely or unlikely to be taken. Many of today's compilers already provide mechanisms like this, so `[[likely]]` and `[[unlikely]]` standardize these features across compilers.[28]

26. "Optimizing Compiler." Accessed February 6, 2021. https://en.wikipedia.org/wiki/Optimizing_compiler#Specific_techniques.

27. We searched for insights as to why and how you'd use this feature to produce better optimized code. There is little information at this point other than the proposal document "Attributes for Likely and Unlikely Statements (Revision 2)" (https://wg21.link/p0479r2). Section 3, "Motivation and Scope," suggests who should use these features and what they should be used for.

28. Herb Sutter. "Trip Report: Winter ISO C Standards Meeting (Jacksonville)." Sutter's Mill, April 3, 2018. https://herbsutter.com/2018/04/. Herb Sutter is the Convener of the ISO C++ committee and a Software Architect at Microsoft.

To use these, place `[[likely]]` or `[[unlikely]]` before an `if`'s or `else`'s body:

```
if (condition) [[likely]] {
   // statements
}
else {
   // statements
}
```

or in a `switch` 's `case` label:

```
switch (controllingExpression) {
   case 7:
      // statements
      break;
   case 11: [[likely]]
      // statements
```

```
      break;
   default:
      // statements
      break;
}
```

Perf There are subtle issues when using these attributes. Using too many `[[likely]]` and `[[unlikely]]` attributes in your code could actually reduce performance.[29] The document that proposed adding these to the language says for each, "This attribute is intended for specialized optimizations which are implementation-specific. General usage of this attribute is discouraged."[30] For other subtleties, see the proposal document at:

29. Section 9.12.6, "Working Draft, Standard for Programming Language C++," ISO/IEC, April 3, 2020, https://github.com/cplusplus/draft/releases/download/n4861/n4861.pdf.
30. "Attributes for Likely and Unlikely Statements (Revision 2)," https://wg21.link/p0479r2, Section VIII, "Technical Specifications."

```
https://wg21.link/p0479r2
```

If you're working on systems with strict performance requirements, you may want to investigate these attributes further.

23 20.15 A Look Toward C++23

We presented examples of and, in some cases, simply mentioned several possible features for C++23 and beyond throughout this book, including the `std::mdarray` container (Section 6.13), contracts (Section 12.13), ranges enhancements (Section 14.10), a modularized standard library (Section 16.12), concurrent data structures (Section 17.15), parallel ranges algorithms (Section 17.15) and executors (Sections 18.5 and 18.8).

In this section, we overview other features that are being considered for C++23.[31,32] The links we provide to C++ Standards Committee papers in this section are forwarding links that will bring you to the most up-to-date version of each paper. The following `deitel.com` link will forward you to the C++ Standard Committee GitHub site's list of C++23 features currently in progress:

31. Bryce Adelstein Lelbach, Fabio Fracassi, Ben Craig, Billy Baker, Nevin Liber, Ville Voutilainen and Inbal Levi, "Library Evolution Plan for Completing C++23," November 9, 2021. Accessed March 17, 2022. https://wg21.link/p2489. [*Note:* This link will forward to the most recent version of this paper when a newer version is published.]
32. "C++23," Wikipedia. Wikimedia Foundation. Accessed March 17, 2022. `https://en.wikipedia.org/wiki/C%2B%2B23`.

```
https://deitel.com/CPlusPlus23OpenIssues
```

You can track the latest C++23 standard draft efforts at:

```
https://eel.is/c++draft/
```

and

```
https://github.com/cplusplus/draft/
```

Also, check out the C++23 Wikipedia page, which lists many smaller tweaks to the language and libraries:

```
https://en.wikipedia.org/wiki/C%2B%2B23
```

23 Features Likely to Be Included in C++23

- **Asynchronous execution capabilities**: Since *C++20 for Programmers* was sent to publication, the C++ Standards Committee has released an extensive updated proposal for asynchronous-execution capabilities—"P2300R4: `std::execution`" (January 18, 2022). The proposed new features include three kinds of components—**schedulers**, **senders**, and **receivers**—and **customizable asynchronous algorithms**. The concept of an executor has been replaced by **schedulers** and **senders**. For the current version of this paper, visit https://wg21.link/P2300. P2300R4 was the most recent version of the paper at the time of this writing.
- **C++23 Ranges**: Since *C++20 for Programmers* was sent to publication, an updated version of the paper "P2214: A Plan for C++23 Ranges" was released (February 18, 2022). For the current version of this paper, visit https://wg21.link/P2214. P2214R2 was the most recent version of the paper at the time of this writing.
- **`std::generator`** and **`std::lazy`**: In Section 18.4, we created a generator coroutine using Sy Brand's **`generator`** library. The papers "`std::generator`: Synchronous Coroutine Generator for Ranges" and "Add lazy coroutine (coroutine task) type" propose C++ standard library capabilities for generator functions and generalized lazy-evaluation capabilities. For the current versions of these papers, visit https://wg21.link/P2168 and https://wg21.link/p1056, respectively.
- 20 **Formatted output**: We used C++20 text formatting throughout the book, mostly via the open-source library `{fmt}` because our preferred compilers did not yet support the standard **`<format>` header** and the **`std::format` function**. Chapter 19 presented many more C++20 text formatting features using the standard capabilities, which were added to Visual C++ as we sent *C++20 for Programmers* to publication. The paper "Formatted output" proposes adding standard I/O and better Unicode support. For the current version of this paper, visit https://wg21.link/p2093.
- **MDSPAN**: The paper "MDSPAN" proposes a **`std::mdspan`** for creating **views into multidimensional arrays** and for slicing `mdspans`—that is, creating **sub-views**. For the current version of this paper, visit https://wg21.link/p0009.
- **Extended floating-point types**: The paper "P1467R8: Extended floating-point types and standard names" proposes allowing C++ implementations to provide additional floating-point types for other well-known floating-point layouts and defines how such types should interact with both the existing floating-point types and the additional ones defined by the implementation. For the current version of this paper, visit https://wg21.link/p1467.

23 Other Features Being Considered for C++23

- **Associative container enhancements**: The paper "P2363: Extending associative containers with the remaining heterogeneous overloads" proposes additional member function overloads to improve the performance of certain existing operations. For the current version of this paper, visit https://wg21.link/P2363.

- **Comparing smart pointers to raw pointers**: The paper“P2249: Mixed comparisons for smart pointers” proposes allowing smart and raw pointers to be compared for equality to determine whether they point to the same object. For the current version of this paper, visit https://wg21.link/P2249.
- **Additional `constexpr` enhancements in the standard library**: The paper “P2283: `constexpr` for specialized memory algorithms” proposes adding constexpr to various standard library algorithms in header `<memory>`. For the current version of this paper, visit https://wg21.link/P2283.
- **Passing list initializers to standard algorithms**: In Section 20.13, we demonstrated creating a function that could receive a list initializer as an argument. The paper “P2248: Enabling List-Initialization For Algorithms” proposes new standard library algorithm overloads that accept initializer lists as arguments. For the current version of this paper, visit https://wg21.link/P2248.
- **Memory allocator enhancements**: The papers “Allocator-aware library wrappers for dynamic allocation” and “`allocate_unique` and `allocator_delete`” propose enhancements that enable developers to customize allocators with more aspects of the standard library—currently, you can do this only for container elements and `shared_ptr`s. For the current version of these papers, visit https://wg21.link/P0211 and https://wg21.link/P0316.
- **`std::span` enhancements**: The paper “P2447: `std::span` And The Missing Constructor” proposes fixes for various scenarios in which you might expect to be able to use a `std::span`, but which currently produce compilation errors. For the current version of this paper, visit https://wg21.link/P2447.“
- **`std::function_ref` enhancements**: The paper "P2472: Make `function_ref` More Functional” proposes enhancements that make references to functions easier to use. For the current version of this paper, visit https://wg21.link/P2472.

20.16 Wrap-Up

In this chapter, we overviewed miscellaneous C++ topics beyond what we were able to include in the print book. We used the `const_cast` operator to “cast away” the `const`-ness of an object. We discussed storage classes, storage duration and objects lifetimes. Then, we used the `mutable` storage-class specifier to indicate that a data member should always be modifiable, even when it appears in an object that’s currently being treated as a `const`.

We showed how to use `namespace`s to avoid naming conflicts. You saw that some operator symbols can be represented as operator keywords. We also showed the mechanics of the `->*` and `.*` pointer-to-member operators.

Next, we showed how to use `dynamic_cast` to safely downcast a base-class pointer to a derived-class pointer so you could access derived-class-specific capabilities when polymorphically processing objects related by a class hierarchy. In that example, we also demonstrated runtime type information (RTTI) for dynamically discovering an object’s type, and we highlighted the C++ Core Guidelines’ cautions regarding hierarchy navigation with `dynamic_cast`.

We showed how to inherit base-class constructors for cases in which a derived class’s constructors have the same parameters as the base class’s constructor and simply forward their arguments to the corresponding base-class constructors.

17 20 We presented C++17’s `[[nodiscard]]` attribute for indicating that a function’s return value should not be ignored and used C++20’s `[[nodiscard]]` enhancement to specify a reason why the return value should not be ignored. The compiler displays the reason in a warning message if you do not use a `[[nodiscard]]` function’s return value.

11 We showed how to use the C++11 smart pointer standard library class templates `shared_ptr` and `weak_ptr` for managing shared resources. We demonstrated how to use custom deleter functions to allow `shared_ptr`s to manage resources that require special destruction procedures. We also explained how `weak_ptr`s can be used to prevent memory leaks in circularly referential data.

11 11 We discussed trailing return types for functions and showed that C++11's `decltype` operator enables the compiler to determine an expression's type at compile time. Both of these capabilities are used extensively in template metaprogramming. Next, you learned how to use the C++11 class template `initializer_list` to create a function that can be called with a braced initializer.

20 23 We presented C++20 attributes `[likely]]` and `[[unlikely]]` for indicating which branches of `if`...`else` or `switch` statements are more or less likely to execute, so the compiler can optimize the code accordingly. Finally, we presented features being considered for C++23.

20.17 Closing Notes

We will regularly update online Chapters 19 and 20—especially Chapter 20's look ahead to C++23. We are currently writing **C++23 for Programmers** and hope to publish it as close as possible to when C++23 is approved by the ISO C++ Standards Committee. If you subscribe to O'Reilly Online Learning (https://learning.oreilly.com), we'll be adding C++23 "Rough Cut" content to our **C++20 for Programmers** e-book and "Sneak Peek" video content to our **C++20 Fundamentals LiveLessons**. We'd be grateful for your comments, criticisms and suggestions for improving this evolving new C++23 material!

A. Operator Precedence and Grouping

Operators are shown in decreasing order of precedence from top to bottom.

Operator	Type	Grouping
`::`	scope resolution	left to right
`()`	grouping parentheses	
`()`	function call	left to right
`[]`	array subscript	
`.`	member selection via object	
`->`	member selection via pointer	
`++`	unary postfix increment	
`--`	unary postfix decrement	
`typeid`	runtime type information	
`dynamic_cast`<*type*>	runtime type-checked cast	
`static_cast`<*type*>	compile-time type-checked cast	
`reinterpret_cast`<*type*>	cast for nonstandard conversions	
`const_cast`<*type*>	cast away `const`-ness	
`++`	unary prefix increment	right to left
`--`	unary prefix decrement	
`+`	unary plus	
`-`	unary minus	
`!`	unary logical negation	

Operator	Type	Grouping
`~`	unary bitwise complement	
`sizeof`	determine size in bytes	
&	address	
`*`	dereference	
`new`	dynamic memory allocation	
`new[]`	dynamic array allocation	
`delete`	dynamic memory deallocation	
`delete[]`	dynamic array deallocation	
`co_await`	await completion of a coroutine	
(*type*)	C-style unary cast	
`.*`	pointer to member via object	left to right
`->*`	pointer to member via pointer	
`*`	multiplication	left to right
`/`	division	
`%`	remainder	
`+`	addition	left to right
`-`	subtraction	
`<<`	bitwise left shift	left to right
`>>`	bitwise right shift	
`<=>`	three-way comparison	left to right

Operator	Type	Grouping
`<`	relational less than	left to right
`<=`	relational less than or equal to	
`>`	relational greater than	
`>=`	relational greater than or equal to	
`==`	relational is equal to	left to right
`!=`	relational is not equal to	
`&`	bitwise AND	left to right
`^`	bitwise exclusive OR	left to right
`|`	bitwise inclusive OR	left to right
`&&`	logical AND	left to right
`||`	logical OR	left to right
`?:`	ternary conditional	right to left
`=`	assignment	
`+=`	addition assignment	
`-=`	subtraction assignment	
`*=`	multiplication assignment	
`/=`	division assignment	
`%=`	remainder assignment	
`&=`	bitwise AND assignment	
`^=`	bitwise exclusive OR assignment	

Operator	Type	Grouping
`|=`	bitwise inclusive OR assignment	
`<<=`	bitwise left-shift assignment	
`>>=`	bitwise right-shift assignment	
`co_yield`	suspend coroutine and return a value	
`throw`	throw an exception	
`,`	comma	left to right

B. Character Set

ASCII Character Set

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

The digits at the left of the table are the left digits of the decimal equivalents (0–127) of the character codes, and the digits at the top of the table are the right digits of the character codes. For example, the character code for “F” is 70, and the character code for “&” is 38.

C. Number Systems

Objectives

In this appendix, you'll:

- Become familiar with number systems concepts, such as base, positional value and symbol value.
- Work with numbers in the binary, octal and hexadecimal number systems.
- Abbreviate binary numbers as octal or hexadecimal numbers.
- Convert octal and hexadecimal numbers to binary numbers.
- Convert back and forth between decimal numbers and their binary, octal and hexadecimal equivalents.
- Understand binary arithmetic and how negative binary numbers are represented using two's complement notation.

Outline

C.1 Introduction
C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers
C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers
C.4 Converting from Binary, Octal or Hexadecimal to Decimal
C.5 Converting from Decimal to Binary, Octal or Hexadecimal
C.6 Negative Binary Numbers: Two's Complement Notation

C.1 Introduction

This appendix introduces the number systems that programmers commonly use, especially when working on software projects that require close interaction with machine-level hardware. Projects like this include operating systems, computer networking software, compilers, database systems and applications requiring high performance.

Decimal Number System

By default, integers such as 227 or –63 use the **decimal (base 10) number system**. The digits in a decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. The lowest digit is 0, and the highest is 9—one less than the base of 10.

Binary Number System

Internally, computers use the **binary (base 2) number system**. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0, and its highest is 1—one less than the base of 2.

As you'll see, binary numbers tend to be much longer than their decimal equivalents. Programmers who work in assembly languages and high-level languages that enable them to reach down to the machine level find it cumbersome to work with binary numbers. So two other number

systems—the octal number system (base 8) and the hexadecimal number system (base 16)—are popular because they make it convenient to abbreviate binary numbers.

Octal Number System

In the **octal number system**, the digits range from 0 to 7. Because both the binary and octal number systems have fewer digits than the decimal number system, their digits are the same as those in decimal.

Hexadecimal Number System

The **hexadecimal number system** poses a problem because it requires 16 digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15. Thus in hexadecimal, we can have numbers like 876 consisting solely of decimal-like digits, numbers like 8A55F consisting of digits and letters and numbers like FFE consisting solely of letters. Occasionally, a hexadecimal number spells a common word such as FACE or FEED—this can appear strange to programmers accustomed to working with numbers.

Hexadecimal Number System

The two tables below summarize the digits of the binary, octal, decimal and hexadecimal number systems.

Binary digit	Octal digit	Decimal digit	Hexadecimal digit
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8

Binary digit	Octal digit	Decimal digit	Hexadecimal digit
		9	9
			A (decimal value of 10)
			B (decimal value of 11)
			C (decimal value of 12)
			D (decimal value of 13)
			E (decimal value of 14)
			F (decimal value of 15)

Attribute	Binary	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Lowest digit	0	0	0	0
Highest digit	1	7	9	F

Positional Notation in the Decimal Number System

Each number system uses **positional notation** in which each digit position has a different **positional value**. For example, in the decimal number 937 shown in the following table, we say that the 7 is written in the ones position, the 3 is written in the tens position, and the 9 is written in the hundreds position. The 9, the 3 and the 7 are referred to as **symbol values**. Each position is a power of the base—base 10 for the decimal number system. The powers begin with 0 and increase by 1 as we move left in the number.

Positional values in the decimal number system			
Decimal digit	9	3	7
Position name	Hundreds	Tens	Ones
Positional value	100	10	1
Positional value as a power of the base (10)	10^2	10^1	10^0

For longer decimal numbers, the next positions to the left are the thousands position (10 to the 3rd power), the ten-thousands position (10 to the 4th power), the hundred-thousands position (10 to the 5th power), the millions position (10 to the 6th power), the ten-millions position (10 to the 7th power) and so on.

Positional Notation in the Binary Number System

In the binary number 101, the rightmost 1 is in the ones position, the 0 is in the twos position, and the leftmost 1 is in the fours position. Each position is a power of the base (base 2), and the powers begin at 0 and increase by 1 as we move left in the number as shown in the following table. So, the binary value 101 can be transformed to decimal with the expression $(1 * 2^2) + (0 * 2^1) + (1 * 2^0)$, which is 4 + 0 + 1. So binary 101 in decimal is 5.

Positional values in the binary number system			
Binary digit	1	0	1
Position name	Fours	Twos	Ones
Positional value	4	2	1
Positional value as a power of the base (2)	2^2	2^1	2^0

For longer binary numbers, the next positions to the left are the eights position (2^3), the sixteens position (2^4), the thirty-twos position (2^5), the sixty-fours position (2^6) and so on.

Positional Notation in the Octal Number System

In the octal number 425, we say that the 5 is written in the ones position, the 2 is written in the eights position, and the 4 is written in the sixty-fours position. Each position is a power of the base (base 8), and the powers begin at 0 and increase by 1 as we move left in the number as shown in the following table.

Positional values in the octal number system			
Octal digit	4	2	5
Position name	Sixty-fours	Eights	Ones
Positional value	64	8	1
Positional value as a power of the base (8)	8^2	8^1	8^0

For longer octal numbers, the next positions to the left are the five-hundred-and-twelves position (8^3), the four-thousand-and-ninety-sixes position (8^4), the thirty-two-thousand-seven-hundred-and-sixty-eights position (8^5) and so on.

Positional Notation in the Hexadecimal Number System

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position, and the 3 is written in the two-hundred-and-fifty-sixes position. Each position is a power of the base (base 16), and the powers begin at 0 and increase by 1 as we move left in the number (Fig. C.6).

Positional values in the hexadecimal number system			
Hexadecimal digit	3	D	A
Position name	Two-hundred-and-fifty-sixes	Sixteens	Ones
Positional value	256	16	1
Positional value as a power of the base (16)	16^2	16^1	16^0

For longer hexadecimal numbers, the next positions to the left would be the four-thousand-and-ninety-sixes position (16^3), the sixty-five-thousand-five-hundred-and-thirty-sixes position (16^4) and so on.

C.2 Abbreviating Binary Numbers as Octal and Hexadecimal Numbers

In computing, octal and hexadecimal numbers are used to abbreviate lengthy binary values. The following table shows that binary values can be expressed concisely in number systems with higher bases than the binary number system.

Decimal number	Binary representation	Octal representation	Hexadecimal representation
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3

Decimal number	Binary representation	Octal representation	Hexadecimal representation
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
1000	1111101000	1750	3E8
10000	10011100010000	23420	2710
100000	11000011010100000	303240	186A0
1000000	11110100001001000000	3641100	F4240

An important relationship that the octal and hexadecimal number systems have to the binary system is that the octal and hexadecimal bases, 8 and 16, are powers of the binary number system's base 2. Consider the following 12-digit binary number and its octal and hexadecimal equivalents.

Binary number	Octal equivalent	Hexadecimal equivalent
100011010001	4321	8D1

Can you determine how this relationship makes it convenient to abbreviate binary numbers in octal or hexadecimal? The answers follow below.

Converting Binary to Octal

To see how the binary number converts easily to octal, simply break the 12-digit binary number into groups of three consecutive bits each, starting from the right. Write those groups over the corresponding digits of the octal number as follows:

100	011	010	001
4	3	2	1

The octal digit under each group of three bits corresponds precisely to the octal equivalent of that 3-digit binary number, as shown in the previous table.

Converting Binary to Hexadecimal

Similarly, to convert from binary to hexadecimal, break the 12-digit binary number into groups of four consecutive bits each, starting from the right. Write those groups over the corresponding digits of the hexadecimal number as follows:

1000	1101	0001
8	D	1

The hexadecimal digit under each group of four bits corresponds precisely to the hexadecimal equivalent of that 4-digit binary number, as shown in the previous table.

C.3 Converting Octal and Hexadecimal Numbers to Binary Numbers

You convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and rewriting them as their equivalent octal or hexadecimal digits. This process may be used in reverse to produce the binary equivalent of an octal or hexadecimal number.

You can convert the octal number 653 to binary by writing the 6 as its three-digit binary equivalent 110, the 5 as its three-digit binary equivalent 101 and the 3 as its three-digit binary equivalent 011 to form the nine-digit binary number 110101011.

You can convert the hexadecimal number FAD5 to binary by writing the F as its four-digit binary equivalent 1111, the A as its four-digit binary equivalent 1010, the D as its four-digit binary equivalent 1101 and the 5 as its four-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

C.4 Converting from Binary, Octal or Hexadecimal to Decimal

We are accustomed to working in decimal. Therefore it’s often convenient to convert a binary, octal, or hexadecimal number to decimal to get a sense of what the number is “really” worth. Our tables

in Section C.1 express the positional values in decimal. To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its positional value and sum these products. For example, the binary number 110101 is converted to decimal 53, as shown in the following table.

Converting a binary number to decimal						
Positional values:	32	16	8	4	2	1
Symbol values:	1	1	0	1	0	1
Products:	1*32=32	1*16=16	0*8=0	1*4=4	0*2=0	1*1=1
Sum:	= 32 + 16 + 0 + 4 + 0 + 1 = 53					

To convert octal 7614 to decimal 3980, we use the same technique, this time with appropriate octal positional values, as shown in the following table.

Converting an octal number to decimal				
Positional values:	512	64	8	1
Symbol values:	7	6	1	4
Products	7*512=3584	6*64=384	1*8=8	4*1=4
Sum:	= 3584 + 384 + 8 + 4 = 3980			

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time with appropriate hexadecimal positional values, as shown in the following table.

Converting a hexadecimal number to decimal				
Positional values:	4096	256	16	1
Symbol values:	A	D	3	B
Products	A*4096	D*256	3*16=48	B*1=11
	= 10*4096	= 13*256		
	= 40960	= 3328		

Converting a hexadecimal number to decimal	
Sum:	= 40960 + 3328 + 48 + 11 = 44347

C.5 Converting from Decimal to Binary, Octal or Hexadecimal

The conversions in Section C.4 follow naturally from the positional notation conventions. Converting from decimal to binary, octal, or hexadecimal also follows these conventions.

Converting from Decimal to Binary

Suppose we wish to convert decimal 57 to binary. We begin by writing the positional values of the columns right-to-left until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	64	32	16	8	4	2	1

Then, we discard the column with positional value 64, leaving:

Positional values:	32	16	8	4	2	1

Next, we work from the leftmost column to the right. We divide 32 into 57 and observe there is one 32 in 57 with a remainder of 25, so we write 1 in the 32 column. We divide 16 into 25 and observe there is one 16 in 25 with a remainder of 9 and write 1 in the 16 column. We divide 8 into 9 and observe there is one 8 in 9 with a remainder of 1. The next two columns each produce quotients of 0 when their positional values are divided into 1, so we write 0s in the 4 and 2 columns. Finally, 1 into 1 is 1, so we write 1 in the 1 column. This yields:

Positional values:	32	16	8	4	2	1
Symbol values:	1	1	1	0	0	1

and thus decimal 57 is equivalent to binary 111001.

Converting from Decimal to Octal

To convert decimal 103 to octal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	512	64	8	1

Then we discard the column with positional value 512, yielding:

Positional values:	64	8	1

Next, we work from the leftmost column to the right. We divide 64 into 103 and observe there is one 64 in 103 with a remainder of 39, so we write 1 in the 64 column. We divide 8 into 39 and observe there are four 8s in 39 with a remainder of 7 and write 4 in the 8 column. Finally, we divide 1 into 7 and observe there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Positional values:	64	8	1
Symbol values:	1	4	7

and thus decimal 103 is equivalent to octal 147.

Converting from Decimal to Hexadecimal

To convert decimal 375 to hexadecimal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values:	4096	256	16	1

Then we discard the column with positional value 4096, yielding:

Positional values:	256	16	1

Next, we work from the leftmost column to the right. We divide 256 into 375 and observe there is one 256 in 375 with a remainder of 119, so we write 1 in the 256 column. We divide 16 into 119 and observe there are seven 16s in 119 with a remainder of 7 and write 7 in the 16 column. Finally, we divide 1 into 7 and observe there are seven 1s in 7 with no remainder, so we write 7 in the 1 column. This yields:

Positional values:	256	16	1
Symbol values:	1	7	7

and thus decimal 375 is equivalent to hexadecimal 177.

C.6 Negative Binary Numbers: Two’s Complement Notation

The discussion so far in this appendix has focused on positive numbers. This section shows how computers represent negative numbers using **two’s complement notation**. First, we explain how the two’s complement of a binary number is formed. Then, we show why it represents the negative value of the given binary number.

Consider a machine with 32-bit integers. Suppose an `int` variable `value` contains `13`. The 32-bit representation of `value` is

```
00000000 00000000 00000000 00001101
```

To form the negative of `value`, we first form its **one's complement** by applying the **bitwise complement operator (~):**

```
onesComplementOfValue = ~value;
```

Internally, `~value` is `value` with each of its bits reversed—ones become zeros and zeros become ones, as follows:

```
value:

00000000 00000000 00000000 00001101



~value (i.e., value's one's complement):

11111111 11111111 11111111 11110010
```

To form the two's complement of `value`, we simply add 1 to `value`'s one's complement. Thus
Two's complement of `value`:

```
11111111 11111111 11111111 11110011
```

Now, if this is equal to –13, we should be able to add it to binary 13 and obtain a result of 0. Let's try this:

```
 00000000 00000000 00000000 00001101

+11111111 11111111 11111111 11110011

-----------------------------------

 00000000 00000000 00000000 00000000
```

The **carry bit** from the leftmost column is discarded, and we indeed get 0 as a result. If we add the one's complement of a number to the number, the result will be all 1s. The key to getting a result of all zeros is that the two's complement is one more than the one's complement. Addition 1 causes each

column to add to 0 with a carry of 1. The carry keeps moving leftward until it is discarded from the leftmost bit. Thus, the resulting number is all zeros.

Computers actually perform a subtraction, such as

```
x = a - value;
```

by adding the two's complement of `value` to `a`, as follows:

```
x = a + (~value + 1);
```

Suppose `a` is 27 and `value` is 13 as before. If the two's complement of `value` is actually the negative of `value`, then adding the two's complement of value to `a` should produce the result 14. Let's try this:

```
a (i.e., 27)   00000000 00000000 00000000 00011011

+(~value + 1) +11111111 11111111 11111111 11110011

               -----------------------------------

               00000000 00000000 00000000 00001110
```

which is indeed equal to 14.

D. Preprocessor

Objectives

In this appendix, you'll:

- Understand `#include` in the context of developing large programs.
- Understand include guards for ensuring a header is included only once per translation unit.
- Use `#define` to create macros and macros with arguments.
- Understand conditional compilation.
- Display error messages during conditional compilation.
- Use assertions to test if the values of expressions are correct.

Outline

D.1 Introduction
D.2 `#include` Preprocessing Directive
D.3 `#define` Preprocessing Directive: Symbolic Constants
D.4 `#define` Preprocessing Directive: Macros
D.5 Conditional Compilation
D.6 `#error` and `#pragma` Preprocessing Directives
D.7 Operators # and ##
D.8 Predefined Symbolic Constants
D.9 Assertions

D.1 Introduction

20 This appendix discusses preprocessor directives in more depth. We provide it primarily for programmers handling legacy C++ code. C++20 modules (Chapter 16) and other Modern C++ techniques such as `constexpr` and templates are preferred to using the preprocessor directives shown here.

Some preprocessor actions are:

- including headers into C++ source-code files,
- defining **symbolic constants** and **macros**,
- **conditionally compiling** source code, and
- **conditionally executing preprocessing directives.**

Err All preprocessing directives begin with `#`, and only whitespace characters may appear before a preprocessing directive on a line. Preprocessing directives are not C++ statements, so they do not end in a semicolon (;). They are processed fully for a translation unit before it is compiled. Placing a semicolon at the end of a preprocessing directive can lead to various errors.

D.2 `#include` Preprocessing Directive

20 The **`#include` preprocessing directive** has been used throughout this text. It includes the text contents of a specified file in place of the directive. With C++20 modules, you can now `import` many headers as header units. Some compilers also enable you to `import` a modular version of the entire

standard library or specific portions of it. As we discussed in Chapter 16, this can significantly reduce compile times and translation unit sizes.

The two forms of the `#include` directive are

```
#include <filename>
#include "filename"
```

The difference is the location the preprocessor searches for the included file. For angle brackets (`<` and `>`), the preprocessor searches implementation-dependent predesignated folders and folders you add to the preprocessor's search path. For quotes, the preprocessor searches first in the same directory as the file being preprocessed, then in the same folders as files contained in angle brackets. Quotes typically are used to include programmer-defined header files.

D.3 `#define` Preprocessing Directive: Symbolic Constants

The **`#define` preprocessing directive** creates

- **symbolic constants**—constants represented as symbols—and
- macros (Section D.4)—function-like operations defined as symbols.

The C++ standard refers to both symbolic constants and macros as macros. The `#define` preprocessing directive format is

```
#define identifier replacement-text
```

The preprocessor replaces all subsequent occurrences of *identifier* (except those in string literals) in the file with *replacement-text* before the program is compiled. After encountering

```
#define PI 3.14159
```

the preprocessor replaces all subsequent occurrences `PI` with `3.14159`.

Everything to the right of the symbolic constant name replaces the symbolic constant. For example, if you were to accidentally include an `=`, as in

```
#define PI = 3.14159
```

Err the preprocessor to replace every occurrence of `PI` with `"= 3.14159"`. Replacements like this cause many subtle logic and syntax errors. Redefining a symbolic constant with a new value is also an error.

CG Err The C++ Core Guidelines say not to use the preprocessor for text manipulations like `PI` shown above. They indicate that "macros are a major source of bugs" and "don't obey the usual scope and type rules."[1,2] Instead, you should prefer `const` and `constexpr` variables, which have a specific data type and are visible by name to a debugger. Once a symbolic constant is replaced with its replacement text, only the replacement text is visible to a debugger.

1. C++ Core Guidelines, "ES.30: Don't use macros for program text manipulation." Accessed March 8, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-macros.
2. C++ Core Guidelines, "Don't use macros for constants or 'functions'." Accessed March 8, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-macros2.

D.4 `#define` Preprocessing Directive: Macros

This section is included for the benefit of C++ programmers who will need to work with C legacy code. Rather than macros, Modern C++ programs should use templates and functions.

You can define function-like macros in `#define` preprocessing directives. As with a symbolic constant, the preprocessor replaces a *macro-identifier* with its *replacement-text* before the

translation unit is compiled. A macro without arguments is processed like a symbolic constant. In a macro with arguments, the preprocessor substitutes the arguments in the *replacement-text*, then expands the macro—that is, the *replacement-text* replaces the macro-identifier and argument list in the program. **There is no data type checking for macro arguments—a macro is used simply for text substitution.**

Macro for a Circle's Area

Consider the following macro definition with one argument for the area of a circle:

```
#define CIRCLE_AREA(x) (PI * (x) * (x))
```

In the macro call `CIRCLE_AREA(y)`, the preprocessor substitutes `y`'s value for `x` in the replacement text, inserts the symbolic constant `PI`'s value (defined previously) and expands the macro in the program. For example, the statement

```
area = CIRCLE_AREA(4);
```

expands `CIRCLE_AREA(4)`, as in

```
area = (3.14159 * (4) * (4));
```

Because the expression consists only of constants, the compiler can evaluate the expression and the result will be assigned to `area` at runtime. The parentheses around each `x` in the replacement text and the entire expression force the proper order of evaluation when the macro argument is an expression. For example, the statement

```
area = CIRCLE_AREA(c + 2);
```

expands `CIRCLE_AREA(c + 2)`, as in

```
area = (3.14159 * (c + 2) * (c + 2));
```

This evaluates correctly because the parentheses force the proper order of evaluation. If the parentheses are omitted, the macro expansion becomes

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly as

```
area = (3.14159 * c) + (2 * c) + 2;
```

Err CG because of the rules of operator precedence. Forgetting to enclose macro arguments in parentheses in the replacement text is an error. Like symbolic constants, macros are error-prone, and the C++ Core Guidelines say to avoid them.[3]

3. C++ Core Guidelines, "Don't use macros for constants or 'functions'." Accessed March 8, 2022. https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-macros2.

Function for a Circle's Area

Macro `CIRCLE_AREA` should be defined as a function, as in

```
constexpr double circleArea(double x) {return 3.14159 * x * x;}
```

Pref If you require support for multiple data types, define `circleArea` as a function template instead. Though there's overhead associated with a function call, modern C++ compilers often can perform optimizations to eliminate that overhead, and `constexpr` functions, in particular, can be completely evaluated at compile-time if their arguments are compile-time constants.

Macro for a Rectangle's Area

The following is a macro definition with two arguments for the area of a rectangle:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Wherever `RECTANGLE_AREA( a, b )` appears in the program, the values of `a` and `b` are substituted in the macro replacement text, and the macro is expanded in place of the macro name. For example, the statement

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

is expanded to

```
rectArea = ((a + 4) * (b + 7));
```

The value of the expression is evaluated and assigned to variable `rectArea`.

Defining Multiline Macros

The replacement text for a macro or symbolic constant usually consists of any text to the right of the identifier and its argument list in the `#define` directive. If the replacement text for a macro or symbolic constant is longer than the remainder of the line, you must place a backslash (\) at the end of each line of the macro except the last. These indicate that the replacement text continues on the next line.

Undefining Macros

Symbolic constants and macros can be discarded using the **#undefpreprocessing directive**. Directive `#undef` "undefines" a symbolic constant or macro name. The scope of a symbolic constant or macro is from its definition until it is either undefined with `#undef` or the end of the file is reached. Once undefined, a name can be redefined with `#define`.

Avoid Expressions with Side Effects When Calling Macros

Err Note that expressions with side effects (e.g., variable values are modified) should not be passed to a macro because **macro arguments might be evaluated more than once**. Macros can accidentally replace identifiers that were not intended to be a use of the macro but just happened to be spelled the same. This can lead to exceptionally mysterious compilation and syntax errors. On the other hand, if you define the same identifier more than once in C++ code, you'll get a compilation error.

D.5 Conditional Compilation

Conditional compilation enables you to control the execution of preprocessing directives and the compilation of program code. Each of the conditional preprocessing directives evaluates a constant integer expression that will determine whether the code will be compiled. Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessing directives because these are all determined by the compiler and preprocessing happens before compilation.

The conditional preprocessor construct is much like the `if` selection structure. Consider the following preprocessor code:

```
#ifndef NULL
   #define NULL 0
#endif
```

which determines whether the symbolic constant `NULL` is already defined. The expression `#ifndef NULL` includes the code up to `#endif` if `NULL` is not defined and skips the code if `NULL` is defined. Every **`#if`** construct must end with **`#endif`**. The directives **`#ifdef`** and **`#ifndef`** are shorthand for `#if defined`(*name*) and `#if !defined`(*name*). A multiple-part conditional preprocessor construct may be tested using the `#elif` (the equivalent of `else if` in an `if` statement) and the `#else` (the equivalent of `else` in an `if` statement) directives.

"Commenting Out" Large Blocks of Code

During program development, programmers often find it helpful to "comment out" large portions of code to prevent it from being compiled. If the code contains `/*` and `*/` multiline comments, `/*` and `*/` cannot be used because they cannot be nested. Instead, you can use the following preprocessor construct

```
#if 0
   code prevented from compiling
#endif:
```

To enable the code to be compiled, simply replace the value `0` in the preceding construct with the value `1`.

Conditional Compilation in Debugging

Conditional compilation is commonly used as a debugging aid. Output statements are often used to print variable values and confirm the flow of control. These output statements can be enclosed in conditional preprocessing directives so that the statements are compiled only until the debugging process is completed. For example,

```
#ifdef DEBUG
   cerr << "Variable x = " << x << "\n";
#endif
```

causes the `cerr` statement to be compiled in the program if the symbolic constant `DEBUG` has been defined before directive `#ifdef DEBUG`. This symbolic constant is normally set by a command-line compiler or by settings in the IDE (e.g., Visual Studio) and not by an explicit `#define` definition. When debugging is completed, the `#define` directive is removed from the source file, and the output statements inserted for debugging purposes are ignored during compilation. In larger programs, it might be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.

Inserting conditionally compiled output statements for debugging purposes in locations where C++ currently expects a single statement can lead to syntax errors and logic errors. In this case, the conditionally compiled statement should be enclosed in a compound statement. Thus, when the program is compiled with debugging statements, the flow of control of the program is not altered.

Include Guards

In our headers, we use

```
#pragma once
```

to ensure that their contents are included into a given translation unit only once. Though this directive is nonstandard, it's widely supported by popular C++ compilers, including all the compilers we use in this book.

The standard way to prevent multiple inclusion is with an **`#include` guard**, which consists of conditional compilation and `#define` directives, as in:

```
#ifndef HEADER_NAME
#define HEADER_NAME
   ...
#endif
```

where `HEADER_NAME` is a symbolic constant that, by convention, uses the header name in uppercase with the period replaced by an underscore.

The `#include` guard prevents the code between `#ifndef` and `#endif` from being `#included` if `HEADER_NAME` has been defined. When a header containing an `#include` guard is `#included` the first time, the identifier `HEADER_NAME` is not yet defined. In this case, the `#define` directive defines `HEADER_NAME`, and the preprocessor includes the header's contents in the translation unit. If the header is `#included`

again, `HEADER_NAME` already would be defined, so any code between `#ifndef` and `#endif` would be ignored.

Attempts to include a header multiple times (inadvertently) often occur in large programs with many headers that, in turn, include other headers. This could lead to compilation errors if the same definition appears more than once in a preprocessed file. Chapter 16 discussed how C++20 modules help prevent such problems.

D.6 `#error` and `#pragma` Preprocessing Directives

The `#error` directive

```
#error tokens
```

prints an implementation-dependent message including the *tokens* specified in the directive. The tokens are sequences of characters separated by spaces. For example,

```
#error 1 - Out of range error
```

contains six tokens. In one popular C++ compiler, for example, when an `#error` directive is processed, the tokens in the directive are displayed as an error message, preprocessing stops, and the program does not compile.

The `#pragma` directive

```
#pragma tokens
```

causes an implementation-defined action. A `pragma` not recognized by the implementation is ignored. A particular C++ compiler, for example, might recognize `pragma`s that enable you to take advantage of that compiler's specific capabilities.

D.7 Operators # and

The `#` and `##` preprocessor operators are available in C++ and ANSI/ISO C. The `#` operator causes a replacement-text token to be converted to a string surrounded by quotes. Consider the following macro definition:

```
#define HELLO (x) cout << "Hello, " #x << "\n";
```

When `HELLO(John)` appears in a program file, it is expanded to

```
cout << "Hello, " "John" << "\n";
```

The string `"John"` replaces `#x` in the replacement text. Strings separated by white space are concatenated during preprocessing, so the above statement is equivalent to

```
cout << "Hello, John" << "\n";
```

The `#` operator must be used in a macro with arguments because the operand of `#` refers to a macro argument.

The `##` operator concatenates two tokens. Consider the following macro definition:

```
#define TOKENCONCAT (x, y) x ## y
```

When `TOKENCONCAT` appears in the program, its arguments are concatenated and used to replace the macro. For example, `TOKENCONCAT(O, K)` is replaced by `OK` in the program. The `##` operator must have two operands.

D.8 Predefined Symbolic Constants

The following table shows several **predefined symbolic constants**. The identifiers for all but `__cplusplus` begin and end with two underscores. These identifiers and preprocessor operator `defined` (Section D.5) cannot be used in `#define` or `#undef` directives. For the complete list, see https://en.cppreference.com/w/cpp/preprocessor/replace.

Symbolic constant	Description
`__LINE__`	The line number of the current source-code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form `"Mmm dd yyyy"` such as `"Aug 19 2002"`).
`__STDC__`	Indicates whether the program conforms to the ANSI/ISO C standard. Contains value 1 if there is full conformance and is undefined otherwise.
`__TIME__`	The time the source file is compiled (a string literal of the form `"hh:mm:ss"`).
`__cplusplus`	Contains the value `199711L` (until C++11), `201103L` (C++11), `201402L` (C++14), `201703L` (C++17) or `202002L` (C++20)

D.9 Assertions

The **`assert`macro**—defined in the **`<cassert>`** header file—tests the value of an expression. If the value of the expression is `0` (false), then `assert` prints an error message and calls function **`abort`** (of the general utilities library—`<cstdlib>`) to terminate program execution. This is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable `x` should never be larger than `10` in a program. An assertion may be used to test the value of `x` and print an error message if the value of `x` is incorrect. The statement would be

```
assert(x <= 10);
```

If `x` is greater than `10`, an error message containing the line number and file name is displayed, and the program terminates. You would then concentrate on this area of the code to find the error. If the symbolic constant `NDEBUG` is defined, subsequent assertions will be ignored. Thus, when assertions are no longer needed (i.e., when debugging is complete), we insert the line

```
#define NDEBUG
```

in the program file rather than deleting each assertion manually. As with the `DEBUG` symbolic constant, `NDEBUG` is often set by compiler command-line options or through a setting in the IDE.

E. Bit Manipulation

Outline

E.1 Introduction
E.2 Bitwise Operators
E.3 Bit Fields

E.1 Introduction

This appendix discusses the manipulation of bits in integer data. We discuss the **bitwise operators** that allow you to access and manipulate the individual bits in bytes of data. We also present **bit fields**—structures that can be used to specify the exact number of bits a variable occupies in memory.

C++ provides extensive **bit-manipulation** capabilities for getting down to the "bits-and-bytes" hardware level. Bit manipulation is typically used in programs that need to communicate directly with hardware. It's also used in encryption data compression and other algorithms.[1] We introduce each **bitwise operator** and discuss how to save memory by using **bit fields**.

---

1. "Bit manipulation." Wikipedia. Wikimedia Foundation. Accessed March 30, 2022. https://en.wikipedia.org/wiki/Bit_manipulation.

E.2 Bitwise Operators

All data is represented internally by computers as sequences of bits. Each bit can assume the value `0` or the value `1`. On most systems, a sequence of eight bits forms a **byte**—the standard storage unit for a variable of type `char`. Other data types are stored in larger numbers of bytes. Bitwise operators manipulate the bits of integral operands (`char`, `short`, `int` and `long`; both signed and `unsigned`)—typically, **unsigned integers.**

The bitwise operator discussions in this section show the binary representations of the integer operands. For a detailed explanation of the binary (base-2) number system, see online Appendix C. Some of these programs might not work on your system without modification due to the **machine-dependent nature of bitwise manipulations**.

The bitwise operators are:

- **bitwise AND (`&`)**,
- **bitwise inclusive OR (`|`)**,
- **bitwise exclusive OR (`^`)**,
- **left shift (`<<`)**,
- **right shift (`>>`)** and
- **bitwise complement (`~`)**—also known as the **one's complement**.

We've been using `&`, `<<` and `>>` for other purposes—this is a classic example of operator overloading. Detailed discussions of each bitwise operator appear in the following examples. The bitwise operators are summarized in the following table.

Operator	Name	Description
`&`	bitwise AND	The bits in the result are set to `1` if the corresponding bits in the two operands are both `1`.
`|`	bitwise inclusive OR	The bits in the result are set to `1` if one or both of the corresponding bits in the two operands is `1`.
`^`	bitwise exclusive OR	The bits in the result are set to `1` if exactly one of the corresponding bits in the two operands is `1`.
`<<`	left shift	Shifts the bits of the first operand left by the number of bits specified by the second operand. Fills from the right with `0` bits.
`>>`	right shift with sign extension	Shifts the bits of the first operand right by the number of bits specified by the second operand. The method of filling from the left is machine-dependent.
`~`	bitwise complement	All `0` bits are set to `1`, and all `1` bits are set to `0`.

Printing a Binary Representation of an Integral Value

When using the bitwise operators, it's useful to illustrate their precise effects by printing values in their binary representation. The program of Fig. E.1 prints an `unsigned` integer in its binary representation in groups of eight bits each.

```
1  // figE_01.cpp
2  // Printing an unsigned integer in bits.
3  #include <fmt/format.h>
4  #include <iostream>
5
6  void displayBits(unsigned); // prototype
7
8  int main() {
9     unsigned inputValue{0}; // integral value to print in binary
10
11    std::cout << "Enter an unsigned integer: ";
12    std::cin >> inputValue;
13    displayBits(inputValue);
14 }
15
16 // display bits of an unsigned integer value
17 void displayBits(unsigned value) {
18    const unsigned SHIFT{8 * sizeof(unsigned) - 1};
19    const unsigned MASK{static_cast<const unsigned>(1 << SHIFT)};
20
```

```
21      std::cout << fmt::format("{:10d} = ", value);
22
23      // display bits
24      for (unsigned i{1}; i <= SHIFT + 1; ++i) {
25         std::cout << (value & MASK ? '1' : '0');
26         value <<= 1; // shift value left by 1
27
28         if (i % 8 == 0) { // output a space after 8 bits
29            std::cout << ' ';
30         }
31      }
32
33      std::cout << "\n";
34   }
```

```
Enter an unsigned integer: 65000
     65000 = 00000000 00000000 11111101 11101000
Enter an unsigned integer: 29
     29 = 00000000 00000000 00000000 00011101
```

Fig. E.1 Printing an unsigned integer in bits.

Function `displayBits` (lines 17–34) uses the **bitwise AND** operator to combine variable `value` with constant `MASK`. The **bitwise AND** operator is often used with an operand called a **mask**—an integer value with specific bits set to `1`. Masks hide some bits in a value while selecting other bits. In `displayBits`, line 19 initializes constant `MASK` with `1 << SHIFT`. The value of constant `SHIFT` was calculated in line 18 by

```
8 * sizeof(unsigned) - 1
```

This expression multiplies an `unsigned` object's number of bytes by `8` (the number of bits in a byte) to get the total number of bits required to store an `unsigned` object, then subtracts 1. The bit representation of `1 << SHIFT` on a computer that represents `unsigned` objects in four bytes of memory is

```
10000000 00000000 00000000 00000000
```

The **left-shift operator** shifts the value `1` from the low-order (rightmost) bit to the high-order (leftmost) bit in `MASK` and fills with `0` bits from the right. Line 25 prints a `1` or a `0` for the current leftmost bit of `value`.

Assume `value` contains `65000`, which in bits is

```
00000000 00000000 11111101 11101000
```

When you combine `value` and `MASK` using `&`, all the bits except the high-order bit in `value` are "masked off" (hidden) because any bit "ANDed" with `0` yields `0`.

If the leftmost bit is `1`, `value & MASK` evaluates to

```
00000000 00000000 11111101 11101000    (value)
10000000 00000000 00000000 00000000    (MASK)
---------------------------------
```

```
00000000 00000000 00000000 00000000    (value & MASK)
```

The value `0` is treated as `false`, and `displayBits` displays a `0`. Then line 26 shifts `value` left by one bit with the expression `value <<= 1`. These steps are repeated for each bit in `value`. Eventually, a `1` bit shifts into the leftmost bit position. Consider the following bitwise AND:

```
11111101 11101000 00000000 00000000    (value)
10000000 00000000 00000000 00000000    (MASK)
---------------------------------
10000000 00000000 00000000 00000000    (value & MASK)
```

Both left bits are `1`s, so the `&` expression's result is nonzero (`true`), and `displayBits` displays a `1`. The following table summarizes combining two bits with the bitwise AND (`&`) operator.

Bit 1	Bit 2	Bit 1 & Bit 2
0	0	0
1	0	0
0	1	0
1	1	1

The program of Fig. E.2 demonstrates the **bitwise AND operator**, the **bitwise inclusive OR** operator, the **bitwise exclusive OR** operator and the **bitwise complement operator**. Function `displayBits` is identical to the one in Fig. E.1.

```
1   // figE_02.cpp
2   // Bitwise AND, inclusive OR,
3   // exclusive OR and complement operators.
4   #include <fmt/format.h>
5   #include <iostream>
6
7   void displayBits(unsigned); // prototype
8
9   int main() {
10     // demonstrate bitwise &
11     unsigned number1{2179876355};
12     unsigned mask{1};
13     std::cout << "The result of combining the following\n";
14     displayBits(number1);
15     displayBits(mask);
16     std::cout << "using the bitwise AND operator & is\n";
17     displayBits(number1 & mask);
18
19     // demonstrate bitwise |
20     number1 = 15;
21     unsigned setBits{241};
22     std::cout << "\nThe result of combining the following\n";
23     displayBits(number1);
24     displayBits(setBits);
25     std::cout << "using the bitwise inclusive OR operator | is\n";
```

```
26       displayBits(number1 | setBits);
27
28       // demonstrate bitwise exclusive OR
29       number1 = 139;
30       unsigned number2{199};
31       std::cout << "\nThe result of combining the following\n";
32       displayBits(number1);
33       displayBits(number2);
34       std::cout << "using the bitwise exclusive OR operator ^ is\n";
35       displayBits(number1 ^ number2);
36
37       // demonstrate bitwise complement
38       number1 = 21845;
39       std::cout << "\nThe one's complement of\n";
40       displayBits(number1);
41       std::cout << "is\n";
42       displayBits(~number1);
43    }
44
45    // display bits of an unsigned integer value
46    void displayBits(unsigned value) {
47       const unsigned SHIFT{8 * sizeof(unsigned) - 1};
48       const unsigned MASK{static_cast<const unsigned>(1 << SHIFT)};
49
50       std::cout << fmt::format("{:10d} = ", value);
51
52       // display bits
53       for (unsigned i{1}; i <= SHIFT + 1; ++i) {
54          std::cout << (value & MASK ? '1': '0');
55          value <<= 1; // shift value left by 1
56
57          if (i % 8 == 0) { // output a space after 8 bits
58             std::cout << ' ';
59          }
60       }
61
62       std::cout << "\n";
63    }
```

```
The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001

The result of combining the following
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       255 = 00000000 00000000 00000000 11111111

The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
        76 = 00000000 00000000 00000000 01001100
```

```
The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

Fig. E.2 Bitwise AND, inclusive OR, exclusive OR and complement operators.

Bitwise AND Operator (&)

In Fig. E.2, line 11 uses `2179876355` (`10000001 11101110 01000110 00000011`) to initialize variable `number1`, and line 12 uses `1` (`00000000 00000000 00000000 00000001`) to initialize variable `mask`. When `mask` and `number1` are combined using the **bitwise AND operator (&)** in the expression `number1 & mask` (line 17), the result is `00000000 00000000 00000000 00000001`. All the bits except the low-order bit in variable `number1` are "masked off" (hidden) by "ANDing" with constant `MASK`.

Bitwise Inclusive OR Operator (|)

The **bitwise inclusive OR operator** sets specific bits to 1 in an operand. In Fig. E.2, line 20 assigns `15` (`00000000 00000000 00000000 00001111`) to variable `number1`, and line 21 uses `241` (`00000000 00000000 00000000 11110001`) to initialize variable `setBits`. When `number1` and `setBits` are combined using the **bitwise inclusive OR operator (|)** in the expression `number1 | setBits` (line 16), the result is `255` (`00000000 00000000 00000000 11111111`). The following table summarizes combining two bits with the **bitwise inclusive-OR operator**.

Bit 1	Bit 2	Bit 1 | Bit 2
0	0	0
1	0	1
0	1	1
1	1	1

Bitwise Exclusive OR (^)

The **bitwise exclusive OR operator** (`^`) sets each bit in the result to 1 if exactly one of the corresponding bits in its two operands is 1. In Fig. E.2, lines 29–30 give variables `number1` and `number2` the values `139` (`00000000 00000000 00000000 10001011`) and `199` (`00000000 00000000 00000000 11000111`), respectively. When these variables are combined with the **bitwise exclusive OR operator** in the expression `number1 ^ number2` (line 35), the result is `00000000 00000000 00000000 01001100`. The following table summarizes combining two bits with the **bitwise exclusive OR operator**.

Bit 1	Bit 2	Bit 1 ^ Bit 2
0	0	0
1	0	1
0	1	1

Bit 1	Bit 2	Bit 1 | Bit 2
1	1	0

Bitwise Complement (~)

The **bitwise complement operator** (~) sets all 1 bits in its operand to 0 in the result and all 0 bits to 1. This is referred to as "taking the **one's complement** of the value." In Fig. E.2, line 38 assigns variable number1 the value 21845 (00000000 00000000 01010101 01010101). When the expression ~number1 evaluates, the result is (11111111 11111111 10101010 10101010).

Bitwise Shift Operators

Figure E.3 demonstrates the **left-shift** (<<) and the **right-shift> (>>)operators**. Function displayBits is identical to the one in Fig. E.1.

```
1  // figE_3.cpp
2  // Using the bitwise shift operators.
3  #include <fmt/format.h>
4  #include <iostream>
5
6  void displayBits(unsigned); // prototype
7
8  int main() {
9     unsigned number1{960};
10
11    // demonstrate bitwise left shift
12    std::cout << "The result of left shifting\n";
13    displayBits(number1);
14    std::cout << "8 bit positions using the left-shift operator is\n";
15    displayBits(number1 << 8);
16
17    // demonstrate bitwise right shift
18    std::cout << "\nThe result of right shifting\n";
19    displayBits(number1);
20    std::cout << "8 bit positions using the right-shift operator is\n";
21    displayBits(number1 >> 8);
22 }
23
24 // display bits of an unsigned integer value
25 void displayBits(unsigned value) {
26    const unsigned SHIFT{8 * sizeof(unsigned) - 1};
27    const unsigned MASK{static_cast<const unsigned>(1 << SHIFT)};
28
29    std::cout << fmt::format("{:10d} = ", value);
30
31    // display bits
32    for (unsigned i{1}; i <= SHIFT + 1; ++i) {
33       std::cout << (value & MASK ? '1' : '0');
34       value <<= 1; // shift value left by 1
35
36       if (i % 8 == 0) { // output a space after 8 bits
```

```
37            std::cout << ' ';
38         }
39      }
40
41      std::cout << "\n";
42   }
```

```
The result of left shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the left-shift operator is
    245760 = 00000000 00000011 11000000 00000000

The result of right shifting
       960 = 00000000 00000000 00000011 11000000
8 bit positions using the right-shift operator is
         3 = 00000000 00000000 00000000 00000011
```

Fig. E.3 Bitwise shift operators.

Left-Shift Operator

The **left-shift operator (`<<`)** shifts the bits of its left operand to the left by the number of bits specified in its right operand. Bits vacated to the right are replaced with `0`s; bits shifted off the left are lost. In Fig. E.3, line 9 initializes variable `number1` with the value `960` (`00000000 00000000 00000011 11000000`). The result of left-shifting `number1` eight bits in the expression `number1 << 8` (line 15) is `245760` (`00000000 00000011 11000000 00000000`).

Right-Shift Operator

The **right-shift operator (`>>`)** shifts the bits of its left operand to the right by the number of bits specified in its right operand. Performing a right shift on an `unsigned` integer causes the vacated bits at the left to be replaced by 0s; bits shifted off the right are lost. In Fig. E.3, the result of right-shifting `number1` in the expression `number1 >> 8` (line 21) is `3` (`00000000 00000000 00000000 00000011`).

Err SE The result of shifting a value is undefined if the right operand is negative or if the right operand is greater than or equal to the number of bits in which the left operand is stored. **The result of right-shifting a signed value is machine-dependent**. Some machines fill with zeros. Others use the sign bit.

Bitwise Assignment Operators

Each bitwise operator (except the bitwise complement operator) has a corresponding assignment operator. These **bitwise assignment operators** are shown in the following table. Each modifies its left operand.

Bitwise assignment operators	
`&=`	Bitwise AND assignment operator.
`|=`	Bitwise inclusive OR assignment operator.
`^=`	Bitwise exclusive OR assignment operator.

Bitwise assignment operators	
`<<=`	Left-shift assignment operator.
`>>=`	Right-shift with sign extension assignment operator.

E.3 Bit Fields

You can specify the number of bits in which an integral type or `enum` type member of a class or a structure is stored. Such a member is referred to as a **bit field** and enables **better memory utilization** by storing data in the minimum number of bits required. Bit field members must have an integral or `enum` type.

The following structure definition contains three `unsigned` bit fields—`face`, `suit` and `color`—used to represent a card from a deck of 52 cards:

```
struct BitCard {
   unsigned face : 4;
   unsigned suit : 2;
   unsigned color : 1;
};
```

You declare a bit field by following an integral type or `enum` type member with a colon (`:`) and an **integer constant** representing the **bit field's width**—the number of bits in which the member is stored.

`BitCard`'s definition indicates that `face` is stored in four bits, `suit` in 2 bits and `color` in one bit. The number of bits for each is based on the member's desired range of values.

- Member `face` stores values between `0` (Ace) and `12` (King)—four bits can store a value between 0 and 15.
- Member `suit` stores values between `0` and `3` (`0` = Diamonds, `1` = Hearts, `2` = Clubs, `3` = Spades)—two bits can store a value between `0` and `3`.
- Member `color` stores either `0` (Red) or `1` (Black)—one bit can store either `0` or `1`.

The program in Figs. E.4–E.6 creates `array deck` containing `BitCard` structures (line 23 of Fig. E.4). The constructor inserts the 52 cards in the `deck array`, and the function `deal` prints the 52 cards. Bit fields are accessed via the dot (`.`) operator (lines 11–13 and 21–26 of Fig. E.5).

```
1   // Fig. E.4: DeckOfCards.h
2   // Definition of class DeckOfCards that
3   // represents a deck of playing cards.
4   #include <array>
5
6   // BitCard structure definition with bit fields
7   struct BitCard {
8      unsigned face : 4; // 4 bits; 0-15
9      unsigned suit : 2; // 2 bits; 0-3
10     unsigned color : 1; // 1 bit; 0-1
11  };
12
13  // DeckOfCards class definition
14  class DeckOfCards {
15  public:
```

```
16      static const int faces{13};
17      static const int colors{2}; // black and red
18      static const int numberOfCards{52};
19
20      DeckOfCards(); // constructor initializes deck
21      void deal() const; // deals cards in deck
22   private:
23      std::array<BitCard, numberOfCards> deck; // represents deck of cards
24   };
```

Fig. E.4 Definition of class `DeckOfCards` that represents a deck of playing cards.

```
1   // Fig. E.5: DeckOfCards.cpp
2   // Member-function definitions for class DeckOfCards that simulates
3   // the shuffling and dealing of a deck of playing cards.
4   #include <fmt/format.h>
5   #include <iostream>
6   #include "DeckOfCards.h" // DeckOfCards class definition
7
8   // no-argument DeckOfCards constructor intializes deck
9   DeckOfCards::DeckOfCards() {
10     for (size_t i{0}; i < deck.size(); ++i) {
11        deck[i].face = i % faces; // faces in order
12        deck[i].suit = i / faces; // suits in order
13        deck[i].color = i / (faces * colors); // colors in order
14     }
15  }
16
17  // deal cards in deck
18  void DeckOfCards::deal() const {
19     for (size_t k1{0}, k2{k1 + deck.size() / 2};
20        k1 < deck.size() / 2 - 1; ++k1, ++k2) {
21        std::cout << fmt::format("Card:{:3d}", deck[k1].face)
22           << fmt::format("  Suit:{:2d}", deck[k1].suit)
23           << fmt::format("  Color:{:2d}", deck[k1].color)
24           << fmt::format("   Card:{:3d}", deck[k2].face)
25           << fmt::format("  Suit:{:2d}", deck[k2].suit)
26           << fmt::format("  Color:{:2d}\n", deck[k2].color);
27     }
28  }
```

Fig. E.5 Member-function definitions for class `DeckOfCards`.

```
1   // figE_06.cpp
2   // Card shuffling and dealing program.
3   #include "DeckOfCards.h" // DeckOfCards class definition
4
5   int main() {
6      DeckOfCards deckOfCards; // create DeckOfCards object
7      deckOfCards.deal(); // deal the cards in the deck
8   }
```

```
Card:  0 Suit: 0 Color: 0  Card:  0 Suit: 2 Color: 1
Card:  1 Suit: 0 Color: 0  Card:  1 Suit: 2 Color: 1
Card:  2 Suit: 0 Color: 0  Card:  2 Suit: 2 Color: 1
Card:  3 Suit: 0 Color: 0  Card:  3 Suit: 2 Color: 1
Card:  4 Suit: 0 Color: 0  Card:  4 Suit: 2 Color: 1
```

```
Card:  5 Suit: 0 Color: 0  Card:  5 Suit: 2 Color: 1
Card:  6 Suit: 0 Color: 0  Card:  6 Suit: 2 Color: 1
Card:  7 Suit: 0 Color: 0  Card:  7 Suit: 2 Color: 1
Card:  8 Suit: 0 Color: 0  Card:  8 Suit: 2 Color: 1
Card:  9 Suit: 0 Color: 0  Card:  9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0  Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0  Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0  Card: 12 Suit: 2 Color: 1
Card:  0 Suit: 1 Color: 0  Card:  0 Suit: 3 Color: 1
Card:  1 Suit: 1 Color: 0  Card:  1 Suit: 3 Color: 1
Card:  2 Suit: 1 Color: 0  Card:  2 Suit: 3 Color: 1
Card:  3 Suit: 1 Color: 0  Card:  3 Suit: 3 Color: 1
Card:  4 Suit: 1 Color: 0  Card:  4 Suit: 3 Color: 1
Card:  5 Suit: 1 Color: 0  Card:  5 Suit: 3 Color: 1
Card:  6 Suit: 1 Color: 0  Card:  6 Suit: 3 Color: 1
Card:  7 Suit: 1 Color: 0  Card:  7 Suit: 3 Color: 1
Card:  8 Suit: 1 Color: 0  Card:  8 Suit: 3 Color: 1
Card:  9 Suit: 1 Color: 0  Card:  9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0  Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0  Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0  Card: 12 Suit: 3 Color: 1
```

Fig. E.6 Bit fields used to store a deck of cards.

Unnamed Bit Fields

An **unnamed bit field** can be used as **padding** in the structure. For example, the structure definition uses an unnamed three-bit field as padding—nothing can be stored in those three bits. Member `b` is stored in another storage unit:

```
struct Example {
   unsigned a : 13;
   unsigned   : 3; // align to next storage-unit boundary
   unsigned b : 4;
};
```

Zero-Width Unnamed Bit Fields

An **unnamed bit field with a zero width** is used to align the next bit field on a new storage-unit boundary. For example, the following structure definition uses an unnamed `0`-bit field to skip the remaining bits (as many as there are) of the storage unit in which `a` is stored and align `b` on the next storage-unit boundary:

```
struct Example {
   unsigned a : 13;
   unsigned   : 0; // align to next storage-unit boundary
   unsigned b : 4;
};
```

Bit Field Notes

When using bit fields, keep the following in mind:

- SE Bit-field manipulations are machine-dependent. Some computers allow bit fields to cross storage unit boundaries, whereas others do not.
- Err Attempting to take the address of a bit field is a compilation error. The `&` operator may not be used with bit fields because a pointer can designate only a particular byte in memory. On the other hand, bit fields can start in the middle of a byte.
- Perf Although bit fields save space, using them can cause the compiler to generate slower-executing machine-language code. This occurs because it takes extra machine-language operations to access only portions of an addressable storage unit.

Index

Symbols

, (comma operator) 75
: in inheritance 345
:: (scope resolution operator) 133, 287, 321
! (logical negation) 88, 90
!= (inequality operator) 31, 32
?: (ternary conditional operator) 47, 145
. (member selection operator) 291, 292
' (digit separator, C++14) 63
'\0' (null character) 216
'\n' (newline character) 216
[] (operator for map) 541
[] (regex character class) 262
[&] (lambda introducer, capture by reference) 562
[=] (lambda introducer, capture by value) 562
{n,} (quantifier in regex) 264
{n,m} (quantifier in regex) 264
* (multiplication operator) 30
* (pointer dereference or indirection operator) 193, 194
* (quantifier in regex) 263
*= (multiplication assignment operator) 57
/ (division operator) 30
/* */ (multiline comment) 23
// (single-line comment) 23
/= (division assignment operator) 57
\ (regex metacharacter) 261
\' (single-quote-character escape sequence) 25
\" (double-quote-character escape sequence) 25
\\ (backslash-character escape sequence) 25
\a (alert escape sequence) 25
\D (regex character class) 262
\d (regex character class) 262, 262
\n (newline escape sequence) 25
\r (carriage-return escape sequence) 25
\S (regex character class) 262
\s (regex character class) 262
\t (tab escape sequence) 25
\W (regex character class) 262
\w (regex character class) 262
& (address operator) 193, 194
& (to declare reference) 129
&& (logical AND operator) 88, 89, 145
&= (bitwise AND assignment operator) 548
% (remainder operator) 30
%= (remainder assignment operator) 57
^ (regex metacharacter) 263
^= (bitwise exclusive OR assignment operator) 549
+ (addition operator) 29, 30
+ (quantifier in regex) 263
-- (postfix decrement operator) 58
++ (postfix increment operator) 58
on an iterator 513
-- (prefix decrement operator) 58
++ (prefix increment operator) 58
on an iterator 513
+= (addition assignment operator) 57
string concatenation 224
< (less-than operator) 31
<< (stream insertion operator) 24, 30
<= (less-than-or-equal-to operator) 31
<=> (three-way comparison operator) 418, 459, 460, 511
= (assignment operator) 29, 30, 304, 424, 513
-= (subtraction assignment operator) 57
= 0 (pure specifier for a pure virtual function) 363
== (equality operator) 31, 32
-> (arrow member selection) 291
> (greater-than operator) 31
-> (in a compound C++20 concept requirement) 656
>= (greater-than-or-equal-to operator) 31
>> (stream extraction operator) 29
| (operator in a C++20 range pipeline) 178
|= (bitwise inclusive OR assignment operator) 548
|| (logical OR operator) 88, 89, 145

A

abbreviated function template (C++20) 137, 634, 634, 650, 651, 662
constrained auto 650
abort standard library function 299, 480, 489
absolute value 104
abstract class 362, 363, 374
Employee 364
pure 363
access a global variable 133
access function 292
access non-static class data members and member functions 324
access privileges 202, 204
access shared data 783
access specifier 274, 275, 313
private 274
public 274
access the caller's data 129
access violation 508
accounts-receivable system 240
accumulate algorithm 174, 596, 605, 608, 621
acquire
a lock 787
a semaphore 827
acquire member function of std::binary_semaphore 829, 830
action expression in the UML 41
action state in the UML 41
symbol 41
activity diagram in the UML 41, 47
activity in the UML 41
ad-hoc constraint (C++20 concepts) 656
adapter 543
add an integer to a pointer 208
adding strings 38
addition 30, 31

compound assignment operator, += 57
address operator (&) 193, 194, 195, 424
adjacent_difference algorithm 621
adjacent_find algorithm 620
ADL (argument-dependent lookup) 637
advance function 652
aggregate initialization 673
aggregate type 324, 328, 670
designated initializer 325
aggregation 308
aiming a derived-class pointer at a base-class object 355
air-traffic-control systems xxi
alert escape sequence ('\a') 25
algebraic expression 30
<algorithm> header 112, 444, 452, 501, 523, 556
algorithms (standard library) 507, 518
accumulate 174, 596, 605
binary_search 168
copy_backward 585
for manipulating containers 103
for_each 455
gcd 596, 596
iota 596, 597
is_sorted 501
iter_swap 582, 583
lcm 596, 597
max 281, 594, 594
min 594, 594
minmax 594, 595
multipass 515
partial_sum 596, 598
reduce 596, 597
separated from container 558
sort 168, 759
specialized memory 621
swap 582, 583
swap_ranges 582
alias 131
declaration (using) 394
for a type 394, 680
for the name of an object 302
all
algorithm 548
range adaptor (C++20) 612
all_of algorithm 620
ranges version (C++20) 578, 580
allocate
memory 112, 425, 425, 427
allocator_type 510
alphanumeric character 262
ambiguity problem 399, 401
angle brackets (< and >) 137
in templates 630
anonymous function 176, 560
any algorithm 548
<any> header 113
any_of algorithm 581, 620
ranges version (C++20) 578, 581
append 224
append data to a file 241
Apple
Xcode xxv, xliii
arbitrary precision integers BigNumber class 61
archive file 94
argument coercion 109
argument-dependent lookup (ADL) 637
arguments in correct order 107
arguments passed to member-object constructors 308
arithmetic
compound assignment operators 57
function object 604
operator xxvi, 30
overflow 492
underflow 492
“arity” of an operator 424
array
built-in 199
C style 190
pointer based 190
array class
template 154, 482, 508
bounds checking 156, 157, 157
container xxvi
multidimensional 170
<array> header 111, 155, 169
to_array function (C++20) 191, 201
array names decay to pointers 199
array subscript operator ([]) 436
arrow member selection operator (->) 291, 292, 316
as_const function (C++17) 676
ASCII (American Standard Code for Information Interchange) character set 85, 216
assert
contract keyword 497
macro 483, 495
macro to disable assertions 483
assertion 483
assign
addresses of base-class and derived-class objects to base-class and derived-class pointers 352
class objects 305
one iterator to another 517
assign member function
list 530
string 224
assignment
operator 57, 304, 424, 513
*= 57
/= 57
%= 57
+= 57
-= 57
= 29
default 304
assignment statement 29
associative
container 516, 533, 535
insert function 534, 538
map 533
multimap 533
multiset 533
ordered 508, 509, 533
set 533
unordered 508, 509, 511, 533
unordered_map 533
unordered_multimap 533
unordered_multiset 533
unordered_set 533
associative container member functions
contains 535
count 534, 540
equal_range 536
extract member function (C++17) 512
find 535
insert 536, 540
lower_bound 536

merge member function (C++17) 533
upper_bound 536
associativity of operators 31
asterisk (*) 30
asynchronous
concurrent threads 782
event 481
programming 103
task 836, 849
task completes 834
at member function 524
array 157, 495
string 224, 422
vector 184, 495
<atomic> header 816
atomic operation 784
atomic pointer 819
atomic types 816
std::atomic class template 817
std::atomic_ref class template (C++20) 817, 820
thread safety 757
atomic_ref class template (C++20) 817, 820
attribute
[[fallthrough]] 83
in the UML 19
of a class 18
of an object 19
audit
contract level 498
level precondition 502
auto keyword 172, 521
automatically destroyed 126
average calculation 48, 49, 50
avoid
naming conflicts 315
protected data 406
repeating code 297
await_ready function of an awaitable object (coroutines) 855
await_resume function of an awaitable object (coroutines) 855
await_suspend function of an awaitable object (coroutines) 855
awaitable object (coroutines) 849, 855
await_ready function 855
await_resume function 855
await_suspend function 855
axiom contract level 498

B

back member function queue 545
sequence containers 524
span class template (C++20) 214
vector 257
back_inserter function template 571, 589
background_executor (concurrencpp) 845
backslash
\ 25
escape sequence, \\ 25
bad data 253, 254
bad_alloc
exception 425, 487, 491
bad_cast exception 491
bad_typeid exception 491
Bancila, Marius blog xxxv
banking systems xxi
bar chart 164
printing program 164
bar of asterisks 164
barrier (C++20) 820, 823
<barrier> header (C++20) 113, 823
base case(s) 140, 144, 146
base class 336, 341
catch 492
default constructor 350
destructor (protected and non-virtual) 377
destructor (public and virtual) 377
exception 491
initializer 345
pointer to a base-class object 352
pointer to a derived-class object 352
private member 405
subobject 402
base *e* 104
base-10 number system 104
basic exception safety guarantee 476
basic searching and sorting algorithms of the standard library 578
basic_ios class 402
basic_iostream class 402
basic_istream class 402
basic_ostream class 402
begin
function 169, 200
header <array> 169
member function of containers 512
member function of first-class containers 513
beginning
of a file 244
of a stream 245
behavior
of a class 18
bell 25
bidirectional
iterator 515, 516, 526, 533, 537, 539, 664
operations 517
bidirectional_iterator concept (C++20) 559, 587
bidirectional_range concept (C++20) 559, 585, 587, 588, 589
big data 222, 250
Big Four C++20
features 628
Big *O* notation 549, 551
BigNumber class 61
pow member function 64
binary function 605
object 605
binary left fold 685, 686, 689
binary operator 29, 30, 90
binary predicate
function 528, 567, 580, 586, 589, 592
binary right fold 686, 689
binary search 500
algorithm 551
binary_search standard library algorithm 168, 170, 620
binary_search standard library algorithm ranges version (C++20) 580
binary tree 508
binary_semaphore (C++20) 827
<bit> header 113
bit manipulation 547
Bitcoin 809
bitset 509, 547

<bitset> header 111
bitwise
assignment operators 548
left-shift operator (<<) 416
operators xxxi
right-shift operator (>>) 416
block 34, 46, 124, 125
of memory 531
scope 124
thread until a lock is
released 787
blockchain 809
blocked thread state 769
blogs
Bancila, Marius xxxv Boccara,
Jonathan xxxv
Filipek, Bartlomiej xxxv
Grimm, Rainer xxxv
Microsoft's C++ Team xxxv
O'Dwyer, Arthur xxxv
Sutter's Mill xxxv
Boccara, Jonathan blog xxxv
body
function 24
if statement 32
Bohm, C. 40
bool
contextual conversion 439
data type 44
bool alpha stream
manipulator 37
Boolean 44
Boolean values in JSON 326
Boost C++ libraries xxiv
Boost.Log logging library 494
Boost.Multiprecision library
(precise floatingpoint
calculations) 75
born thread state 768
bounds checking 157
braced 443
braced initializer 27, 426
list 443
list as constructor argument 443
list for custom classes 443
narrowing conversion 109
braces ({}) 24, 34, 46
not required 83
break statement 83, 86
brittle
base-class problem 406
software 406
broadcast operations 440
buffer 777
buffer overflow 158
build level (contracts) 502
built-in array xxvii, 190, 199

C
C xxxiv
C-like pointer-based array 509
C-string xxvii, 190, 216
C-style arrays 190
C-style string 190
C++
code repositories xxxiv
Language Reference xxxv
open-source community xxxiv
C++ Core
Guidelines xxiii, xxxi, 746
explicit single-parameter
construtor 277
Guidelines Support Library 110
override 361
C++ documentation xxxiv
*C++ How to Program, Eleventh
Edition* xxxvii
C++ language documentation
(Microsoft) xxxv
C++ preprocessor 23
C++ standard
library xxiv, 22, 103
array class template 154
container 154
exception types 491
headers 111
string class 35, 273
<stri ng> header 37
vector class template 181
C++ Standards Committee xxxv
C++11 xxi
auto keyword 172
braced initialization 27
braced initializers as
constructor arguments 443
cend container member
function 522
crbegin container member
function 522
crend container member
function 522
<cstdint> header 207
default special member
function 361, 444
default type arguments for
function template type
parameters 678
delegating constructor 298
fixed-size integer types 207
in-class initializer 285
launch enum 814
list initialization 541
noexcept 448
nullptr constant 192
override 358, 361
<random> header 113
<regex> header 261, 265
scoped enumeration (enum
class) 120
shrink_to_fit container member
function for vector and deque 522
specifying an enum's integral
type 123
static_assert declaration 659
std::async function
template 808, 814
std::begin function 200
std::call_once 816
std::condition_variable class 787
std::condition_variable_any class
805
std::end function 200
std::forward_li st class
template 508
std: future class template 814
std::iota algorithm 621
std::lock_guard class 791
std::minmax algorithm 594, 595
std::move function 438, 439
std::mutex class 787, 788
std::once_flag 816
std::packaged_task function
template 815
std::promise 814
std::random_device random-
number source 118, 123
std::shared_future class
template 815
std: :shared_lock class 804
std::shared_mutex class 804
std: :shared_ptr class
template 428
std::this_thread::get_id function
772
std::this_thread::sleep_for functi
on 773

std::this_thread::sleep_until function 773
std::thread 771
std::to_string function 235
std::unique_lock class 788, 789
std::unique_ptr class template 428, 430
std::unordered_multimap class template 509
std::unordered_multiset class template 509
std::unordered_set class template 509
std::weak_ptr class template 428
stod function 235
stof function 235
stoi function 235
stol function 235
stold function 235
stoll function 235
stoul function 235
stoull function 235
thread_local storage class 758
<tuple> header 111
variadic template 679
C++14 xxi
digit separator ' 63
generic lambdas 176
heterogeneous lookup (associative containers) 537
std::make_unique function template 428, 430
std::quoted stream manipulator 246
string-object literal 420
variable template 678
C++17 xxi
<chrono> header 761
class template argument deduction (CTAD) 158
constexpr if 699
contiguous iterator 515
<execution> header 762
execution policy 762, 763
extract member function of associative containers 512
[[fallthrough]] attribute 83
<filesystem> header 241
fold expression 628, 682
merge member function of associative containers 533
std::as_const function 676
std::exclusive_scan parallel algorithm 766
std::execution::par execution policy 762, 763
std::execution::par_unseq execution policy 763
std::execution::parallel_policy class 763
std::execution::parallel_sequenced_policy class 763
std::execution::seq execution policy 763
std::execution::sequenced_policy class 763
std::execution::unseq execution policy 763
std::execution::unsequenced_policy class 763
std::filesystem::path 241
std::for_each_n parallel algorithm 766
std::inclusive_scan parallel algorithm 766
std::optional class template 191
std::reduce parallel algorithm 766
std::scoped_lock class 791
std::string_view 190, 236, 274
std::transform_exclusive_scan parallel algorithm 766
std::transform_inclusive_scan parallel algorithm 766
std::transform_reduce parallel algorithm 766
<string_view> header 236
structured binding 595
unpack elements via structured binding 577
C++20 xxi
abbreviated function template 634, 634
ad-hoc constraint in concepts 656
<barrier> header 823
bidirectional_iterator concept 587
bidirectional_range concept 585, 587, 588, 589
“big four” features 628
C++ standard document xxxv
co_await operator (coroutines) 834
co_return statement (coroutines) 834, 848
co_yield expression (coroutines) 834, 837, 839
<compare> header 460
concept keyword 648
concepts 411, 556, 558, 636, 640, 652
concepts by header 642
<concepts> header 641
conjunction in a constraint or concept 642
consteval function 699
constrained auto 650
constraint expression in a concept 640, 648
constraint in concepts 640, 641
contiguous_iterator concept 564
contracts (pushed to a later standard) 496
coroutine 834
disjunction in a constraint or concept 642
ends_with member function of class string 38
forward_iterator concept 569, 576
forward_range concept 569, 571, 576, 580, 586, 593
indirectly_copyable concept 561
indirectly_readable concept 561
indirectly_swappable concept 583
indirectly_writable concept 561, 572
input_iterator concept 570, 587
input_or_output_iterator concept 565
input_range concept 561, 566, 567, 568, 570, 572, 573, 574, 575, 577, 578, 579, 580, 581, 582, 583, 584, 586, 587, 589, 590, 591, 592, 595
iterator concepts 559
<latch> header 820
output_iterator concept 564
output_range concept 564
permutable concept 575
projection in a ranges algorithm 567
projection in std::ranges algorithms 608
random_access_iterator concept 575, 579, 600

random_access_range concept 575, 579
range 177, 507
range adaptor 611
range concepts 559
<ranges> header 177
ranges library 177, 253
requires clause 640
requires expression 654
<semaphore> header 827
sentinel of a range 525
standard concepts by header 642
std::all_of algorithm (ranges) 578, 580
std::any_of algorithm (ranges) 578, 581
std::atomic_ref class template 817, 820
std::barrier 820, 823
std::binary_search algorithm (ranges) 578, 580
std::binary_semaphore 827
std::copy algorithm (ranges) 525, 560
std::copy_backward algorithm (ranges) 584, 585
std::copy_if algorithm (ranges) 584, 587
std::copy_n algorithm (ranges) 584, 587
std::count algorithm (ranges) 574, 575, 577
std::count_if algorithm (ranges) 574
std::counting_semaphore 827
std::equal algorithm (ranges) 566, 566
std::equal_range algorithm (ranges) 592
std::fill algorithm (ranges) 563, 564
std::fill_n algorithm (ranges) 563, 564
std::find algorithm (ranges) 578
std::find_if algorithm (ranges) 578, 579
std::find_if_not algorithm (ranges) 578, 582
std::for_each algorithm (ranges) 561
std::format function from header <format> 65, 98
std::generate algorithm (ranges) 563, 564
std::generate_n algorithm (ranges) 563, 565, 565
std::includes algorithm (ranges) 589, 590
std::inplace_merge algorithm (ranges) 588
std::jthread 771, 776
std::latch 820, 820, 821
std::lexicographical_compare algorithm (ranges) 566, 568
std::lower_bound algorithm (ranges) 592, 593
std::make_heap algorithm (ranges) 600
std::max_element algorithm (ranges) 574, 576
std::merge algorithm (ranges) 584, 586
std::min_element algorithm (ranges) 574, 576
std::minmax algorithm (ranges) 595
std::minmax_element algorithm (ranges) 574, 576
std::mismatch algorithm (ranges) 566, 567
std::move algorithm (ranges) 586
std::move_backward algorithm (ranges) 586
std::none_of algorithm (ranges) 578, 581
std::pop_heap algorithm (ranges) 602
std::push_heap algorithm (ranges) 601
std::ranges namespace 525, 560, 561, 563, 566, 568, 572, 574, 578, 582, 584, 588, 589, 592, 594, 599
std::remove algorithm (ranges) 568, 569
std::remove_copy algorithm (ranges) 568, 570
std::remove_copy_if algorithm (ranges) 568, 572
std::remove_if algorithm (ranges) 568, 571
std::replace algorithm (ranges) 572, 572
std::replace_copy algorithm (ranges) 572, 573
std::replace_copy_if algorithm (ranges) 572, 574
std::replace_if algorithm (ranges) 572, 573
std::reverse algorithm (ranges) 584, 587
std::reverse_copy algorithm (ranges) 588, 589
std::same_as concept 649
std::set_difference algorithm (ranges) 589, 591
std::set_intersection algorithm (ranges) 589, 591
std::set_symmetric_-difference algorithm (ranges) 589, 591
std::set_union algorithm (ranges) 592
std::shuffle algorithm (ranges) 574, 575
std::sort algorithm (ranges) 578, 579, 609
std::sort_heap algorithm (ranges) 601
std::span class template of header 191, 210
std::starts_with member function of class string 38
std::stop_callback for cooperative cancellation 808
std::stop_source for cooperative cancellation 807
std::stop_token for cooperative cancellation 807
std::swap_ranges algorithm (ranges) 583, 584
std::to_array function of header <array> 191, 201
std::transform algorithm (ranges) 574
std::unique algorithm (ranges) 584, 586
std::unique_copy algorithm (ranges) 588, 589
std::upper_bound algorithm (ranges) 592, 593
<stop_token> header 805
templated lambda 636
three-way comparison operator (<=>) 460, 511

view 177, 507, 611
viewable_range 611
weakly_incrementable concept 561
C++20 for Programmers code download xliii
C++20 Fundamentals Live-Lessons videos xxxvii
C++20 modules xxiii
transition from the preprocessor 712
C++20 ranges
| operator in a range pipeline 178
pipeline 178
std::views::filter 178, 179
std::views::iota 178
C++23 xxi, 411
concurrent map 831
concurrent queue 830
contracts (could be later than C++23) 496
modular standard library 746
ranges enhancements 622
std::mdarray container 173
C++26 411
Caesar cipher 148
calculations 41
callback function 834
calling functions by reference 195
camel case 28
capacity
of a string 227
of a vector 519
capacity member function
of string 228
of vector 519
capturing variables in a lambda 257, 456
caret (^) regex metacharacter 263
carriage return ('\r') escape sequence 25
cascading
member function calls 316, 317, 319
stream insertion operations 30
case insensitive 266
regular expression 261
case keyword 83
case sensitive 28, 266
regular expression 261
case studies xxv
casino 119
<cassert> header 112, 483
cast operator 52, 210, 463
cast away const-ness 670
overloaded 454
catch block 185
catch exceptions in constructors 484
catch handler 476, 480
all exceptions
with catch(...) 492, 493
base-class exception 492
catch related errors 492
catch(...) (catch all exceptions) 492, 493
cbegin
member function of containers 512
member function of vector 521
<cctype> header 112
ceil function 104
cend
member function of containers 512
member function of vector 522
cereal header-only library 251, 327
JSONInputArchive 331
JSONOutputArchive 329
cerr (standard error stream) 239
<cfloat> header 112
chain of constructor calls 349
chain of destructor calls 350
chaining stream insertion operations 30
char data type 28, 110
character array 216
character class (regular expressions) 261, 262
custom 262
character constant 216
character literal 85, 85
character presentation 112
character sequence 246, 274
character string 24
<chrono> header 111, 284, 761
duration_cast 761
steady_clock 761
cin (standard input stream) 29, 239, 242
cipher
Caesar 148
substitution 148
Vigenère 148, 149, 150
ciphertext 148
circular buffer 795
circular wait (necessary condition for deadlock) 770
clamp algorithm 621
Clang C++ xxiii, xliii, 4
clang++ in a Docker container 4
clang-tidy static analysis tools xxxii, xlviii
class 18
class keyword 137, 273, 273, 630
constructor 275
data member 19
default constructor 278
development 430
diagram in the UML 340
hierarchy 339, 362
implementation programmer 290
interface 284
interface described by function prototypes 107
invariant 295, 495
public services 284
class-average problem 48, 51
class scope 124, 287, 291
static class member 320
class template 155, 409, 627, 629
definition 629
member-function templates 631
scope 634
specialization 629, 630
Stack 630, 632
class template argument deduction (CTAD) 158, 181, 523, 537, 568, 673, 676
class template specialization 155
classes
array class template 154
bitset 509, 547
deque 518, 531
exception 491
forward_list 518
invalid_argument 492
list 518, 526
multimap 539
MyArray 432
numeric_limits 63

out_of_range exception class 185
priority_queue 546, 599, 600, 601
queue 545
runtime_error 472, 480
set 537
stack 543
steady_clock 761
string 35, 273
system_clock 761
tuple 679
unique_ptr 428
vector 180
cleaning data 260
clear member function of containers 512, 526
client
code 351
of a class 279
client-code programmer 290
<climits> header 112
clog (standard error buffered) 239
close member function of ofstream 243
closed set of types 391
cloud 326
cloud-based services 326
cmatch 265
<cmath> header 77, 103, 111
isnan function 256
list of functions 104, 105
mathematical special functions 105
co_await expression (C++20) 849
co_await operator (C++20) 834
co_return statement (C++20) 834, 848
co_yield expression (C++20) 834, 837, 839
code 19
code download xliii
code repositories xxxiv
Coffman, E. G. 770
coin tossing 114
collision in a hashtable 552
colon (:) 399
in inheritance 345
column 170
column headings 157
combining control statements in two ways 92
comma (,) 75
comma operator (,) 75, 145, 690
comma-separated list 27, 34, 75
of base classes 399
of parameters 107
command-line argument 217
Command Prompt window 6
comment 23, 28
multiline 23
single-line 23
CommissionEmployee class
header 369
implementation file 369
test program 343
common programming errors xxiii
common range 524, 557, 560
common range adaptor (C++20) 612
communications systems xxi
CommunityMember class hierarchy 339
commutative operators 459
comparator function object 533, 539
less 533, 546
<compare> header 113, 460
compare iterators 517
compare member function of class string 226
comparing strings 225
compilation error 57
compile 679
compile a header as a header unit 714
compile time calculations 628
compile-time constant 679
polymorphism 408, 410, 513, 628, 629
predicate 640
programs that write code 628
recursion 682, 683
static polymorphism 628
compiler 23, 53
Apple Xcode xliii
Clang C++ xxiii
g++ 11
GNU C++ xxiii, xliii
GNU g++ 4
Microsoft Visual Studio xliii
Visual C++ xxii
Visual Studio Community edition 4
Xcode on macOS 4
Compiler Explorer xxxix
website (godbolt.org) 498
-pthread compiler flag 771
compiler warnings enable 93
completion function 823, 826
barriers 823
component 18
composable 177
composable views 177, 611
composition 308, 311, 337, 341
compound assignment operators 57, 59
compound interest 75
compound requirement in C++20 concepts 654, 655
-> 656
compound statement 34
compression
run-length encoding 94
computing the sum of the elements of an array 163, 174
concatenate 224
stream insertion operations 30
concept-based overloading (C++20) 411, 652, 659, 693, 699
concept overloading 411
concept keyword (C++20) 648
concepts (C++20) 411, 558, 636, 640, 652
-> in a compound requirement 656
ad-hoc constraint 656
bidirectional_iterator 559, 587
bidirectional_range 559, 585, 587, 588, 589
compound requirement 654, 655
concept keyword 648
conjunction 642
constraint 640, 641
constraint expression 640, 648
contiguous_iterator 559, 564
contiguous_range 559
custom 648
disjunction 642
forward_iterator 559, 569, 576
forward_range 559, 569, 571, 576, 580, 586, 593
indirectly_copyable 561

indirectly_readable 561
indirectly_swappable 583
indirectly_writable 561, 572
input_iterator 559, 570, 587, 653
input_or_output_iterator 565
input_range 559, 561, 566, 567, 568, 570, 572, 573, 574, 575, 577, 578, 579, 580, 581, 582, 583, 584, 586, 587, 589, 590, 591, 592, 595
iterators 559
listed by header 642
logical AND (&&) operator in a constraint 642
logical OR (||) operator in a constraint 642
nested requirement 654, 656
output_iterator 559, 564
output_range 559, 564
permutable 575
random_access_iterator 559, 569, 570, 575, 579, 600, 653
random_access_range 559, 575, 579, 600, 601, 602
ranges 559
requires clause 640
requires expression 654
simple requirement 654, 654
standard 640
std::floating_point 641, 648
std::integral 641, 648
std::same_as 649
type requirement 654, 655
weakly_incrementable 561
<concepts> header (C++20) 113, 641
concrete class 362
concrete derived class 365
concurrencpp coroutine support library 836
background_executor 845
executor 836
inline_executor 845, 845
install 837
result 841
runtime 841, 843
submit function of an executor 844
task 836, 841
thread_executor 844
thread_pool_executor 841, 844, 844
timer 836
utility functions 836
when_all function 848
when_any function 849
worker_thread_executor 845
concurrent container
Google Concurrency Library (GCL) 830
Microsoft Parallel Patterns Library 830
concurrent map (C++23) 831
reference implementation 831
concurrent operations 756
concurrent programming 103, 757
with a simple sequential-like coding style 834
concurrent queue (C++23) 830
reference implementations 830
concurrent threads 783
condition 31, 47, 79
Yoda 93
condition variable 789
condition_variable wait function 789
condition_variable class 787
<condition_variable> header (C++11) 112, 787
condition_variable_any class 805
conditional expression 47
conditional operator, ?: 47
confusing equality (==) and assignment (=) operators 92
conjunction in a C++20 constraint or concept 642
const 306
keyword 115
member function 274, 306
member function on a non-const object 307
objects and member functions 307
qualifier 162
qualifier before type specifier in parameter declaration 131
version of operator[] 453
const_cast
cast away const-ness 670
const_iterator 510, 512, 516, 535
const_pointer 510
const_reference 510
const_reverse_iterator 510, 512, 516, 522
constant
compile-time 679
constant integral expression 85
constant pointer
to an integer constant 204
to constant data 202, 204, 205
to nonconstant data 202, 204
constant running time 550
constant variable 162
consteval function (C++20) 699
constexpr function 699
constexpr if (C++17) 699
constexpr qualifier 162, 162
constrained auto (C++20) 650
constraint 640, 648
constraint (C++20 concepts) 640, 641
constraint expression (C++20 concepts) 640, 648
constructor 275, 278
braced-initializer list 443
call chain 349
conversion 462, 464
copy 446
default arguments 296
exception handling 483
explicit 464
function prototype 285
in a class hierarchy 349
injection 386
multiple parameters 280
single argument 464, 465
constructors and destructors called automatically 298
consumer 757, 776
thread 777
container 103, 111, 436, 506, 508
begin function 512
cbegin function 512
cend function 512
clear function 512
crbegin function 512
crend function 512
empty function 512
end function 512
erase function 512
insert function 513
map associative container 533
map class template 509
max_size function 513
multimap associative container 533
multimap class template 509

multiset associative container 533
multiset class template 509
nested type names 672
priority_queue class template 509
queue class template 509
rbegin function 512
rend function 512
sequence 508
set associative container 533
set class template 509
size function 513
special member functions 511
stack class template 509
swap function 513
unordered_map associative container 533
unordered_multimap associative container 533
unordered_multiset associative container 533
unordered_set associative container 533
container (Docker) xxxiv, xlv
container adaptor 508, 509, 509, 516, 543, 543
priority_queue 546, 599, 600, 601
queue 545
stack 543
container adaptor functions
pop 543
push 543
container in the C++ standard library 154
container member function complete list 510
contains function of associative container 535
contextual conversion 454, 466
contextual conversion to bool 439
contiguous iterator (C++17) 515, 558
contiguous_iterator concept (C++20) 559, 564
contiguous_range concept (C++20) 559
continuation mode (for contract violations) 500
continue statement 86, 87
contract 495, 496
assert contract keyword 497
attributes 497
audit contract level 498
axiom contract level 498
build level 502
continuation mode 500
contract_violation 502
default contract level 498
default violation handler 500
design by contract 496
disable contract checking 500
early access implementations (GNU C++) 498
ensures contract keyword 497, 498
expects contract keyword 497, 498
experimental implementation 471
handle_contract_violation default contract violation handler 500
level 498, 502, 503
post contract keyword (GNU C++ early access implementation) 499
pre contract keyword (GNU C++ early access implementation) 499
proposal 497
violation 500, 502
violation handler 503
contract_violation object 502
control statement xxvi, 41, 43
do...while 78, 79
for 42, 71, 72, 75, 77
if 31
nesting 43
stacking 43
switch 80
while 47, 70
control variable 70, 71, 72
controlling expression of a switch 83
converge on the base case 146
conversion, contextual
conversion to bool 439
conversion constructor 418, 462, 464
conversion operator 418, 454, 463
explicit 465
convert among user-defined types and built-in types 463
convert between types 462
convert lowercase letters 112
convert strings to floating-point types 235
convert strings to integral types 235
cooperative 805
cooperative cancellation 776, 805, 807
std::stop_callback 808
std::stop_source 807
std::stop_token 807
cooperative multitasking 835
cooperative thread cancellation 805
coordination types (thread synchronization) 820
copy 472
copy algorithm 441, 444, 523, 620
ranges version (C++20) 525, 560
copy-and-swap idiom 447, 459
strong exception guarantee 477
copy assignment operator (=) xxviii, 278, 417, 431, 446, 513
overloaded 420
copy-constructible type 472
copy constructor xxviii, 278, 306, 311, 417, 421, 431, 434, 437, 445, 446, 511, 513
default 311
copy of the argument 202
copy semantics xxviii, 417, 432
copy_backward algorithm 585, 620
ranges version (C++20) 584, 585
copy_if algorithm 620
ranges version (C++20) 584, 587
copy_n algorithm 620
ranges version (C++20) 584, 587
CopyConstructible 513
coroutine 840
coroutine (C++20) 834
awaitable object 855
co_yield expression 837, 839
coroutine frame 855
coroutine state 855
coroutine support library 835, 849
coroutine_handle 855
generator 837

`generator` coroutine support library (Sy Brand) 837
promise object 854
stackless 840
`suspend_always` 854
`suspend_never` 854
suspension point 855
`<coroutine>` header 113
`<coroutine>` header (C++20) 854
coroutine libraries
`concurrencpp` 836
`cppcoro` 836
`folly::coro` 836
`generator` (Sy Brand) 836, 837
correct number of arguments 107
correct order of arguments 107
`cos` function 104
cosine 104
`count` algorithm 620
ranges version (C++20) 574, 575, 577
`count` function of `multimap` 540
`count` function of associative container 534
`count_down` member function of a `std::latch` 821
`count_if` algorithm 620
ranges version (C++20) 574
`count_if` ranges algorithm (C++20) 257, 258
`counted` range adaptor (C++20) 612
counter 48
counter-controlled iteration xxvi, 48, 48, 52, 70, 71, 146
counting loop 71
`counting_semaphore` (C++20) 827
`cout` (standard output stream) 24, 26, 239
`.cpp` filename extension 719
`cppcheck` static analysis tools xxxii, xlviii
`cppcoro` coroutines library 836
cpplang Slack channel xlvii
`.cppm` filename extension 719
crafting valuable classes with operator overloading 430
craps simulation 119, 120
`crbegin` member function of containers 512
member function of `vector` 522
Create a New Project dialog in Visual Studio Community Edition 5
create a sequential file 240
create an `array` object from a built-in array or an initializer list 201
create an object (instance) 36, 271
create your own data types 30
`CreateAndDestroy` class definition 299
member-function definitions 300
`crend` member function of containers 512
member function of `vector` 522
critical section 784, 787, 788, 795, 827
critical sections 788
cryptocurrency 809
`<cstdint>` header (C++11) 60, 207
`<cstdio>` header 112
`<cstdlib>` header 111, 489, 490
`<cstring>` header 112
CSV (comma-separated values)
`.csv` file extension 250
file format 222, 250
`rapidcsv` header-only library 251
CTAD (class template argument deduction) 158, 523
`<ctime>` header 111
<Ctrl>-d 82, 242
<Ctrl> key 82
<Ctrl>-z 82, 242
curly braces in format string 66
current position in a stream 245
cursor 25
custom character class 262
custom concept 648
custom exception class 472
custom functions xxvi
customization points for derived classes 377

D

dangling pointer 445
dangling reference 131
data mutable 757
data analytics 222, 250
data compression 94
lossless 94
lossy 94
data-interchange format JSON 326
data member 19
data persistence 222
data race 783
data science 222, 250
data structure 154, 506
data types
`char` 110
`float` 110
`int` 27
`long double` 110
`long int` 110
`long long` 110
`long long int` 110
`unsigned char` 110
`unsigned int` 110
`unsigned long` 110
`unsigned long int` 110
`unsigned long long` 110
`unsigned long long int` 110
`unsigned short` 110
`unsigned short int` 110
database 804
dataset 222, 250
Titanic disaster 253
date and time utilities 761
`Date` class 308
dates 103
DbC (design by contract) 496
deadlock 769
four necessary conditions 770
prevention (Havender) 770
process or thread 769
sufficient conditions 770
deallocate memory 425, 427
`Debug` area (Xcode) 9
decay to a pointer (array names) 199
decimal point 53, 54
decision 44
making xxvi
symbol in the UML 44
declaration 27
declarative programming 175
decrement a pointer 208

operator, -- 58, 454
deduction guide 674
deep 445
deep copy 445
deep learning 222, 250
default 311
default arguments 132, 292
with constructors 292
default assignment operator 304
default case in
a switch 83, 84, 117
default
constructor 278, 285, 292, 313, 511
default contract level 498
default copy constructor 311
default destructor 298
default special member
function 361, 444
autogenerate
a virtual destructor 361, 444
default type argument 604
for a type parameter 678, 678
default violation handler
(contracts) 500
default_random_engine 114
#define preprocessing
directive 711
definition 71
Deitel & Associates, Inc. xlii
virtual and on-site corporate
training xlii
Deitel, Dr. Harvey M. xli
Deitel, Paul J. xli
Full-Throttle training
courses xxxvii
Live Instructor-Led
Training xxxvii
delegating
constructor 298
to other functions 632
delete 425, 429
placement 425
delete[] (dynamic array
deallocation) 426
deleting dynamically allocated
memory 427
dependency injection 386
dependent condition 90
deprecated 111
deque class
template 508, 518, 531, 631, 678
push_front function 531
shrink_to_fit member
function 522
<deque> header 111, 531
dereference
a pointer 193, 196, 203
an iterator 513, 515, 517
an iterator positioned outside
its container 522
dereferencing operator (*) 193
derive one class from
another 308
derived class 336, 341
catch 492
customization point 377
pointer to a base-class
object 352
pointer to a derived-class
object 352
descriptive statistics 256, 256
deserialization xxvii
deserializing data 326
design by contract (DbC;
Bertrand Meyer) 496
design pattern 427
design process 20
designated initializer
(aggregates) 325
destructor xxviii, 279, 298, 417,
431, 511
called in reverse order of
constructors 298
in a class hierarchy 350
destructor in a derived class 350
destructors called in reverse
order 350
destructors should not throw
exceptions 483, 486
detach a thread 775
device driver
polymorphism in operating
systems 363
devirtualization 362
diagnostics that aid program
debugging 112
diamond in the UML 41
diamond inheritance (in
multiple inheritance) 402
dice game 119
die rolling
using an array instead
of switch 165
difference_type 510
nested type in an iterator 667
digit 28
digit separator ' (C++14) 63
Dionne, Louis 409
direct access elements of a
container 508
direct base class 340, 340
directly reference a value 192
disable assertions 483
Discord server #include
<C++> xlvii
disjunction in a C++20
constraint or concept 642
disk space 488, 490
dispatch
a thread 768
display a line of text 22
distance algorithm 652
std::ranges 663
distribution (random-number
generation) 114
DivideByZeroException 476
divides function object 604
division 30, 31
by zero 471
compound assignment
operator, /= 57
do...while iteration
statement 42, 78
Docker xxxiv, xlv
Clang C++ container xxiii
Clang container 709
clang++ container 4
container xxxiv, xlv, 709
Docker Desktop 13, 14
Docker Engine 13, 14
GCC Docker container 13
GNU C++ container xxiii
GNU Compiler Collection
(GCC) 13
GNU Compiler Collection (GCC)
container 4, 13
image xlv
Docker Desktop installer xlvi
Docker Hub account xlvi
documentation C++ xxxiv
dot operator
(.) 37, 291, 292, 316, 358, 429
dotted line in the UML 42
double-checked locking 815
double data type 28, 50, 109

double dispatch 411
double-ended queue 531
double-precision floating-point number 77
double quote 25
double-selection statement 42
“doubly initializing” member objects 313
doubly linked list 508, 526
download examples xxii
dreamincode.net/forums/forum/15-c-and-c/ xxxvi
driver program 35
drop range adaptor (C++20) 612, 615
drop_while range adaptor (C++20) 612, 615
dual-core processor 17
duck typing 338, 397, 409
dummy value 50
duplicate keys 533, 539
duration_cast function template 761
dynamic binding 358, 373, 376
dynamic casting 409
dynamic data structure 190
dynamic memory allocation 220, 417, 425, 427, 428, 481, 488
array of integers 441
dynamic_cast 491
dynamically determine function to execute 357, 359
Dyno library for type erasure 409

E

Easylogging++ logging library 494
ECMAScript regular expressions 260
Editor area (Xcode) 8, 9
efficiency of
binary search 551
linear search 550
element of an array 155
elements range adaptor (C++20) 612, 617
else keyword 45
embedded parentheses 31
embedded system xxi, 16, 482
emplace member function of queue 545
of stack 543
emplace_after 512
emplace_front 512
emplace_hint 512
Employee abstract base class 364
Employee class 308
definition showing composition 310
definition with a static data member to track the number of Employee objects in memory 321
header 366
implementation file 366
member-function definitions 310, 322
empty member function of containers 512
of priority_queue 546
of queue 545
of sequence container 525
of stack 543
of string 228
empty member function of string 38, 419
Empty Project template 5
empty statement (a semicolon, ;) 46
empty string 37, 228, 272, 274
enable compiler warnings 93
encapsulation 19, 274, 304
enclosing scope 479
end
function 169, 200
function of header <array> 169
member function of containers 512, 513
end of a stream 245
“end of data entry” 50
end-of-file (EOF) indicator 82, 242
key combination 242
marker 239
ends_with member function of class string (C++20) 38
engine (random-number generation) 114
ensures contract keyword 497, 498
Enter key 29
enum
keyword 123
specifying underlying integral type 123
enum class 120
enumeration 120
constant 120
equal algorithm 441, 452, 620
ranges version (C++20) 566, 566
equal to 31
equal_range
algorithm 620
algorithm, ranges version (C++20) 592
function of associative container 536
equal_to function object 604
equality operators 31, 32
!= 44
== 431
== and != 44
EqualityComparable 513
erase 232
algorithm from header <vector> 570
member function of containers 512
member function of first-class containers 525
member function of string 233
member function of vector 569
erase-remove idiom 569, 569, 570
erase_if algorithm from header <vector> 570
error detected in a constructor 484
error_code class 494
escape character 25
escape sequence 25, 26, 249
\' (single-quote character) 25
\" (double-quote character) 25
\\ (backslash character) 25
\a (alert) 25
\n (newline) 25
\r (carriage return) 25
\t (tab) 25
eText (Pearson) xxxvii
examples (download) xxii
exception 157, 185, 185, 469
bad_alloc 425
handler 185
handling 180
invalid_argument 235

memory footprint 470
out_of_bounds 431
out_of_range 185, 236
parameter 185
what member function of an exception object 186
exception class 491
what virtual function 475
exception guarantee copy-and-swap idiom 477
exception handling 112, 469
flow of control 470, 503
<exception> header 112, 491
exception in a thread 771
exception parameter 474
exception safe code 476
exception safety guarantees
basic exception safety guarantee 476
no guarantee 476
no throw exception safety guarantee 477
strong exception safety guarantee 477
exception types in the C++ standard libarary 491
exceptions
bad_alloc 487
bad_cast 491
bad_typeid 491
filesystem_error 494
length_error 492
logic_error 491
out_of_range 492
overflow_error 492
underflow_error 492
exchange function (header <utility>) 448
exclusive 776
exclusive resource 776
exclusive_scan algorithm 621
parallel algorithm (C++17) 766
execution
parallel 764
<execution> header 113, 762
execution policy
std::execution::par 762
execution policy (C++17) 762, 763
execution-time overhead 373
executor (concurrencpp coroutine support library) 841, 844
scheduling tasks 836
exit a function 25
exit function 242, 299, 299, 489
exit point
of a control statement 43
EXIT_FAILURE 242
EXIT_FAILURE constant 490
EXIT_SUCCESS 242
exiting a for statement 86
exp function 104
expects contract keyword 497, 498
expiring value 438
explicit
conversion 53
narrowing conversion 110
explicit keyword 277, 465
constructor 464
conversion operators 465
exponential “explosion” of calls 145
exponential complexity 145
exponential function 104
exponentiation 77
export (C++20 modules)
a block 716
a declaration 716, 716, 719, 719
a namespace 717
a namespace member 717
export import (C++20 modules) 734
export module (C++20 modules) 718
declaration for a module interface partition 733
expression 44, 53
extensible 337, 351
programming language 270
external iteration 161
extract member function of associative containers (C++17) 512
extracting data from text 260

F

fabs function 104
Facebook
Folly open-source library 409
factorial 61, 140, 141
with partial_sum 599
factorials 599
fail fast 483
[[fallthrough]] attribute 83
false 32, 44, 146
fault-tolerant programs 185, 469
features in a dataset 253
Fertig, Andreas xxxviii
Fibonacci series 143, 145, 836
generator coroutine 837
field width 77, 99
FIFO (first-in, first-out) 509, 531, 545
file 222, 244
file compression ZIP 94
file open mode 241, 243
ios::app 241
ios::ate 241
ios::binary 241
ios::in 241, 243
ios::out 241
ios::trunc 241
file-position pointer 244
file processing 103
file scope 125, 291, 719
filename 241, 243
filename extensions
.cpp 719
.cppm 719
.h 272
.ifc 718
.ixx 718, 718
.pcm 719
<filesystem> header (C++17) 113, 241, 494
filesystem_error 494
filesystem_error class 494
filesystem::path (C++17) 241
Filipek, Bartlomiej blog xxxv
fill algorithm 620
ranges version (C++20) 563, 564
fill_n algorithm 620
ranges version (C++20) 563, 564
filter 611
filter range adaptor (C++20) 612
filtering in functional-style programming 178, 611
final
class 362
member function 361
final state in the UML 41
final value 71

final_suspend function of a coroutine promise object 854
find algorithm 620
ranges version (C++20) 578, 578
find function of associative container 535
find member function of class string 230, 231
find member function of string_view 239
find_end algorithm 620
find_first_not_of member function of class string 232
find_first_of algorithm 620
member function of class string 231
find_if algorithm 620
ranges version (C++20) 578, 579
find_if_not algorithm 620
ranges version (C++20) 578, 582
find_last_of member function of class string 231
finding strings and characters in a string 230
first
data member of pair 536
member function of span class template (C++20) 215
first-class container
begin member function 513
clear function 526
end member function 513
erase function 525
first-in, first-out (FIFO) 509, 531
data struture 545
fixed point
format 53
value 77
fixed-size data structure 199
fixed-size integer types (C++11) 207
fixed stream manipulator 53
flag value 50
float data type 50, 110
floating-point arithmetic 416
floating-point calculations (precise)
Boost.Multiprecision monetary library 75
floating-point division 53
floating-point literal 75, 77
double by default 77
floating-point number 50, 50, 52
double data type 50
double precision 77
float data type 50
single precision 77
floating_point concept 641, 648
floor function 104
flow of control 52
exception handling 470, 503
flow of control in the if...else statement 45
flow of control of a virtual function call 375
fmod function 104
-fmodules-ts compiler flag (g++) 714
{fmt} library (C++20 text formatting) 65, 66, 67
fold expression (C++17) 628, 682, 685
binary left fold 685, 686
binary right fold 686
unary left fold 686
unary right fold 686
fold operation
binary left 689
binary right 689
unary left 688
unary right 688
Folly open-source library from Facebook 409
Poly 409
folly::coro coroutine support library 836
font conventions in this book xxxiii
for iteration statement 42, 71, 72, 75, 77
header 72
for_each algorithm 441, 455, 620
ranges version (C++20) 561
for_each_n algorithm 620
parallel algorithm (C++17) 766
format function from header <format> (C++20) 65, 98
<format> header (C++20) 113
format function 65, 98
format specifier (C++20 text formatting) 99
format string 66, 66
placeholder 66
formatted input/output 246
formatted string curly braces in a replacement field 66
formatted text 246
formatting strings 65
forums
dreamincode.net/forums/forum/15-cand-c/ xxxvi
groups.google.com/g/comp.lang.c++ xxxvi
reddit.com/r/cpp/ xxxvi
stackoverflow.com xxxvi
forward
iterator 515, 527, 558, 583
operations 517
forward_iterator concept (C++20) 559, 569, 576
forward_list class template 508, 518, 527
splice_after member function 528
<forward_list> header 111, 527
forward_range concept (C++20) 559, 569, 571, 576, 580, 586, 593
fragile base-class problem 406
fragile software 406
free function 287, 314, 458
free store 425, 427
friend
of a base class 405
of a derived class 405
friend function 313
can access private members of class 314
friendship granted, not taken 313
friendship
not symmetric 313
not transitive 313
front
member function of queue 545
member function of sequence containers 524
front member function of span class template (C++20) 214
front member function of vector 257
front_inserter function template 571
fstream 240
<fstream> header 112, 240

full template specialization 697
Full-Throttle training courses xxxvii
function xxvi, 18, 24
anonymous 176
call overhead 128
`consteval` 699
`constexpr` 699
definition 125
free 287, 458
header 107
`hypot` 104
overloading 134
parameter 107
parameter list 107
prototype 107, 108, 125, 129
signature 108, 135
that calls itself 139
function call operator `()` 466
function object 533, 533, 539, 553, 557, 603
also called a functor 557, 603
arithmetic 604
binary 605
`divides` 604
`equal_to` 604
`greater` 604
`greater_equal` 604
`less` 604
`less_equal` 604
`less<int>` 533
`less<T>` 539, 546
logical 604
`logical_end` 604
`logical_not` 604
`logical_or` 604
`minus` 604
`modulus` 604
`multiplies` 604
`negate` 604
`not_equal_to` 604
`plus` 604
predefined in the STL 604
relational 604
function overloading 182
function parameter scope 124
function pointer 220, 374, 557, 605
function prototype 107, 313
in a class definition 285
function scope 124
function template 137, 627, 652
abbreviated (C++20) 137, 634, 634
unconstrainted 637
function template specialization 137, 138, 139
function `try` block 484, 485, 486
`<functional>` header 112, 604
functional programming 698
functional structure of a program 24
functional-style programming xxv, 113, 178, 441, 557, 611
filtering 178
mapping 179
reduction 163, 174, 175
functor (function object) 557, 603
fundamental type xxvi, 28, 60
`bool` 44
`char` 28, 110
`double` 50
`float` 50
`int` 57
`long` 60, 60
`long double` 50, 77
`long long` 60, 61
promotion 53
`future` class template
`get` member function 815
`<future>` header 112, 813

G

`g++` compiler 11
in a Docker container 4
game of chance 119
game of craps 120
game playing 113
game systems xxi
garbage value 278
Gates, William xxxiii
GCC Docker container 13
`gcd` algorithm 596, 596, 621
general class average problem 50
generalities 351
generalized numeric operations 619
`generate` algorithm 620
ranges version (C++20) 563, 564
`generate_n` algorithm 620
ranges version (C++20) 563, 565
generating values to be placed into elements of an `array` 162
generator coroutine 837, 839
Fibonacci sequence 837
`generator` coroutine support library 836
Sy Brand 837
`tl::generator` class template 837
generator function 563
generic algorithms 558
generic lambda 176, 177, 201, 395, 562
generic programming xxv, 137
`get` function for obtaining a `tuple` member 681
`get` member function of a `unique_ptr` 444
`get` member function of class template `future` 815
get pointer 244
`get_id` function of the `std::this_thread` namespace (C++11) 772
`get_return_object` function of a coroutine promise object 854
`getline` function of the `string` header 95
gets the value of 32
getting questions answered xxxvi
Git xliii
GitHub xxiv, xxxiii, xxxiv, xxxvi, xxxix, xliii
C++20 Standard Document xxxv
global 287
global function 103
global module 725
global module (C++20 modules) 725
global module fragment 725
global namespace scope 124, 125, 298, 719, 720
global object constructors 298
global scope 298, 300
global variable 125, 125, 125, 128, 133
GNU C++ xxiii, xliii, 4
GNU C++ Standard Library Reference Manual xxxv
GNU Compiler Collection (GCC) Docker container xxv, 4, 13

GNU g++ xxv
GNU GCC 709
Godbolt, Matt
Compiler Explorer xxxix
godbolt.org xxxix, xl
Compiler Explorer web-site 498
Goetz, Brian xl
golden mean (golden ratio) 143
Google C++ Style Guide 493
Google Concurrency Library (GCL) concurrent containers 830
Google Logging Library (glog) 494
Google Search xl
goto elimination 40, 41
goto statement 40
Grammarly xl
graph information 164
greater function object 604
greater_equal function object 604
greater-than operator 31
greater-than-or-equal-to operator 31
greatest common divisor 596
greedy evaluation 611, 836
greedy quantifier 263
Grimm, Rainer xl
blog xxxv
grouping (operators) 31
grouping not changed by overloading 424
groups.google.com/g/comp.lang.c++ xxxvi
<gsl/gsl> header 110
guard condition in the UML 44
guarding code with a lock 788
Guidelines Support Library (GSL) 110, 199

H
.h filename extension (header) 272
half-open range 178, 518
handle on an object 291
handle_contract_violation default contract violation handler 500
has-a relationship 308, 337
hash (hashable keys) 533
hash bucket 552
hash table 552
hash-table collisions 552, 553
hashable 537, 541
type requirements 533
hashing 533, 552
Havender (deadlock prevention) 770
head of a queue 508
header 111, 286, 495
<cstdlib> 490
<gsl/gsl> 110
header (.h) 272, 272
header <numeric> 596
header of a function 107
header-only library 94, 110, 712
inline variable 679
header unit (C++20 modules) 712, 713, 714, 721
compile a header 714
headers 236
<algorithm> 444, 452, 501, 523, 556, 620
<array> 155
<atomic> 816
<barrier> (C++20) 823
<cassert> 483
<chrono> (C++11) 284, 761
<cmath> 77, 103, 104, 105
<compare> 460
<concepts> 641
<condition_variable> (C++11) 787
<coroutine> (C++20) 854
<cstdint> 60
<cstdint> (C++11) 207
<deque> 531
<exception> 491
<execution> (C++17) 762
<filesystem> (C++17) 241, 494
<forward_list> 527
<functional> 604
<future> (C++11) 813
<initializer_list> 443
<iomanip> 53
<iostream> 23
<latch> (C++20) 820
<limits> 63
<list> 526
<map> 539, 541
<memory> 427, 556, 621
<mutex> (C++11) 787, 804
<numbers> 104
<numeric> 556, 621
<queue> 545, 546
<random> (C++11) 113
<ranges> 177
<regex> 261, 265
<semaphore> (C++20) 827
<set> 533
<stack> 543
<stdexcept> 472, 491
<stop_token> (C++20) 805
<string> 37
<thread> (C++11) 771
<tuple> 679
<type_traits> 644, 701
<unordered_map> 539, 541
<unordered_set> 533, 537
<utility> 448
<variant> 391
<vector> 180
heap 546, 599
max heap 599
min heap 599
heapsort
make_heap algorithm 600
pop_heap algorithm 602
push_heap algorithm 601
sort_heap algorithm 601
sorting algorithm 599
helper function 292
heterogeneous lookup (associative containers; C++14) 537
hexadecimal integer 194
hide implementation details 274
hide names in outer scopes 124
hierarchical relationship 339
hierarchy
of employee types 341
of exception classes 490
high-level concurrency features 836
higher-order functions 175
highest level of precedence 31
“highest” type 109
hold-and-wait condition 770
Hollman, Dr. Daisy xxxix
horizontal tab ('\t') 25
HTTPS protocol 148
hypot function 104
hypotenuse
in three-dimensional space 104
of a right triangle 104

I
I/O completion 481

identifier 28, 42, 125
IEEE 754 floating-point standard 500
`if` single-selection statement 31, 42, 44
with initializer 85
`if...else` double-selection statement 42, 44, 45, 52
with initializer 85
`.ifc` filename extension 718
`ifstream` 240, 243, 244
`IGNORECASE` regular expression flag 266
image (Docker) xlv
immutable 174
data 784
data and thread safety 757
keys 509
string literal 217
implement an interface 385
implementation inheritance 349, 391
implementation of a member function changes 297
implicit conversion 53, 277, 463, 464
improper 463
user defined 463
via conversion constructors 464
implicit first argument 315
implicit handle 291
`import` statement (C++20 modules) 720
existing header as a header unit 712
in-class initializer 285, 325
in-memory formatting 247
I/O 247
in parallel 756
include guard 286, 720
`#include <iostream>` 23
`includes` algorithm 620
ranges version (C++20) 589, 590
including a header multiple times 286
`inclusive_scan` algorithm 621
parallel algorithm (C++17) 766
increment 75
a control variable 70, 71
a pointer 208
an iterator 517, 664
expression 87
operator 454
indefinite postponement 768, 769, 770
indentation 45, 46
index 155
indexed access 531
indexed name used as an *rvalue* 436
indirect base class 340, 340
indirection 192
operator (`*`) 193
triple 373
indirectly reference a value 192
`indirectly_copyable` concept (C++20) 561
`indirectly_readable` concept (C++20) 561
`indirectly_swappable` concept (C++20) 583
`indirectly_writable` concept (C++20) 561, 572
inequality operator (`!=`) 431
`-inf` (negative infinity) 471
`inf` (positive infinity) 471
infer (determine) a variable's data type 172
infer a lambda parameter's type 176, 201, 562
infinite loop 73, 74, 140
infinite range 613
infinite sequence 836
information hiding 19, 274
inherit 336
inherit members of an existing class 336
inheritance 19, 308, 336, 337, 340
as an implementation detail 409
hierarchy 339
implementation 391
interface 384, 391
multiple 397, 398, 399
`public` 341
`virtual` 403
initial state in the UML 41
initial value of control variable 70
`initial_suspend` function of a coroutine promise object 854
initialization
once-time, thread-safe 815
`std::call_once` (C++11) 816
`std::once_flag` (C++11) 816
initialize a pointer 192
initializer 158
initializer list 158
`initializer_list` class template 443, 443, 511, 535
`size` member function 443
`<initializer_list>` header 443
initializing
an `array`'s elements to zeros and printing the `array` 156
multidimensional `array`s 170
the elements of an `array` with a declaration 158
`inline` 128
function 128
keyword 128
variable 320, 679
`inline_executor` (concurrencpp) 845
inner block 124
`inner_product` algorithm 621
innermost pair of parentheses 31
`inplace_merge` algorithm 620
ranges version (C++20) 588
input xxvi
input and output stream iterators 514
input from string in memory 112
input iterator 515, 517, 560
input/output (I/O) 103
header `<iostream>` 23
library functions 112
input sequence 514
input stream iterator 514
input stream object (`cin`) 29
`input_iterator` concept (C++20) 559, 570, 587, 653
`input_or_output_iterator` concept (C++20) 565
`input_range` concept (C++20) 559, 561, 566, 567, 568, 570, 572, 573, 574, 575, 577, 578, 579, 580, 581, 582, 583, 584, 586, 587, 589, 590, 591, 592, 595
inputting from `string`s in memory 247
insert 507

at back of vector 518
insert
function of associative container 536
function of containers 513
function of multimap 540
function of multiset 534
function of sequence container 524
function of set 538
function of string 234
inserter function template 571
instance 18
instance variable 273, 281
instantiate
class template 627
template 627
Instructor-Led Training with Paul Deitel xxxvii
int & 129
int data type 24, 28, 57, 109
operands promoted to double 53
integer 24, 27
arithmetic 416
BigNumber class 61
division 30, 50
promotion 53
types, fixed-size 207
integral concept 641, 648
integral constant expression 80, 426
integral expression 85
inter-thread communication 814
std::future 814
std::promise 814
interest rate 75
interface 284, 363, 384
dependency 753
inheritance 364, 384, 391
of a class 284
to a hierarchy 364
internal 175
internal iteration 161
internal linkage 719
International Standards Organization (ISO) 2
invalid_argument exception 235, 287, 492
invariant 495
class 495
invoke function 610
<iomanip> header 53, 111
ios::app file open mode 241
ios::ate file open mode 241
ios::beg seek direction 245
ios::binary file open mode 241
ios::cur seek direction 245
ios::end seek direction 245
ios::in file open mode 241, 243
ios::out file open mode 241
ios::trunc file open mode 241
<iostream> header 23, 111
iota
algorithm 596, 597, 621
range factory (C++20) 613
range factory (C++20), infinite range 613
is-a relationship (inheritance) 337, 407
is_arithmetic type trait 655
is_base_of type trait 699
is_heap algorithm 621
is_heap_until algorithm 621
is_partitioned algorithm 621
is_permutation algorithm 620
is_sorted algorithm 501, 621
is_sorted_until algorithm 621
isEmpty member function of a stack 632
isnan function of header <cmath> 256
ISO (International Standards Organization) 2
Issue navigator 8
istream class 244, 247
seekg function 244
tellg function 245
istream_iterator 514
istringstream class 247, 248, 249
iter_swap algorithm 582, 583, 620
iteration 43, 145, 147
of a loop 87
iteration statement xxvi, 41, 42, 47
do...while 78, 79
while 47, 70
iteration terminates 47
iterative solution 140, 147
factorial 146
iterator 506
contiguous (C++17) 515
minimum requirements 558
nested type names 667, 669
pointing to the first element of the container 513
pointing to the first element past the end of container 513
read/write 668
read-only 665
string 568
type names 516
iterator 510, 512, 513, 516, 536, 538
iterator adaptor 570
back_inserter 571
std::reverse_iterator 672
iterator concepts (C++20) 559
complete list 559
<iterator> header 112, 571
iterator operation 664
iterator_category nested type in an iterator 667
.ixx filename extension 718, 718

J
Jacopini, G. 40
Java Platform Module System (JPMS) 746
join function of a std::jthread 775
joining a thread 844
Joint Strike Fighter Air Vehicle (JSF AV) C++ Coding Standards (2005) 493
JSON (JavaScript Object Notation) 326, 327
array 326
Boolean values 326
cereal library 251
data-interchange format 326
false 326
JSON object 326
null 326
number 326
RapidJSON library 251
serialization 326
string 326
true 326
JSONInputArchive (cereal library) 331
JSONOutputArchive (cereal library) 329
jthread class (C++20) 771, 776

K
Kalev, Danny Ph.D. xxxix
key 533
key–value
pair 509, 539, 540, 542, 552
keyboard 29, 240
keys range adaptor
(C++20) 612, 616
keyword 24, 42
auto 172
break 83
case 83
class 137, 273, 630
co_await (C++20) 834
co_return (C++20) 834
co_yield (C++20) 834
concept (C++20) 648
const 115
constexpr 162
continue 86
default 83
do 42, 78
else 42
enum 123
enum class 120
explicit 277, 464
for 42, 71
if 42
inline 128
namespace 719
operator 423
private 274, 275
public 274
static 123, 124
switch 42
template 630, 630
thread_local (C++11) 758
throw 476
typename 137, 630
unsigned 109
while 42, 78
Kül, Dietmar xxxix, xl

L
label in a switch 83
lambda 176, 557, 560
capture variables 257, 456
expression 176, 455
generic 176, 201, 395
infer a parameter's
type 176, 201, 562
introducer 176, 257, 456, 562
introducer [&] (capture by
reference) 562
introducer [=] (capture by
value) 562
templated (C++20) 636
templatized 634
last-in, first-out (LIFO)
data structure 509, 543
order 629, 632
last member function
of span class template
(C++20) 215
latch (C++20) 820, 820, 820, 821
<latch> header 113
late binding 358
launch a long-running task
asynchronously 834
launch enum 814
async 814
launch policy
(multithreading) 814
lazily computed sequence
(generator) 835
lazy evaluation 178, 611, 836
lazy pipeline 179
lcm algorithm 596, 597, 621
leaf node in a class
hierarchy 364
least common multiple 597
left align (<) in string
formatting 99
left brace ({) 24
left fold
binary 689
unary 688
left justified 45
left-shift operator (<<) 416
left side of an
assignment 93, 155, 302, 436
left stream manipulator 77
left-to-right evaluation 31
left value 93
legacy code 425
length member function of
class string 37
length of a string 217
length_error exception 228, 492
less function object 604
less-than operator 31
less-than-or-equal-to
operator 31
less_equal function object 604
less<int> 533, 539
Levi, Inbal xxxix
lexicographical 226
comparison 37, 419
sort 169
lexicographical_compare algorithm
620
ranges version (C++20) 566, 568
lexicographical_compare_three_way
algorithm 621
libraries
header-only 94
miniz-cpp 94
lifetime of an object 273
LIFO (last-in, first-out) 509, 543
order 632
"light bulb moment" 418
<limits> header 63, 112
linear running time 550
linear search algorithm 551
linked list 507
Linux
shell prompt 4
list class 518, 526
list class template 508
<list> header 111, 526
list member functions
assign 530
merge 529
pop_back 530
pop_front 530
push_front 527
remove 531
sort 528
splice 528
swap 530
unique 530
literal
character 261
digits 261
floating point 77
live-code approach xxii
Live Instructor-Led Training
with Paul Deitel xxxvii
LL for long long integer
literals 63
load factor 552
local automatic object 302
local variable 49, 126, 315
static 125
<locale> header 112
lock

an object 789, 790, 791
release 787
lock_guard class (C++11) 791
log function 104
log10 function 104
logarithm 104
logarithmic running time 551
logging 494
logging libraries
Boost.Log 494
Easylogging++ 494
Google Logging Library (glog) 494
Loguru 494
Plog 494
spdlog 494
logic error 32
slicing 475
logic_error exception 491
logical AND (&&) operator 88, 90
in a constraint 642
logical complement operator, ! 90
logical function object 604
logical negation, ! 90
logical operators xxvi, 88, 90
logical OR (||) operator 88, 89
in a constraint 642
logical_and function object 604
logical_not function object 604
logical_or function object 604
Loguru logging library 494
long data type 60, 60, 110
long double data type 50, 77, 110
long long data type 60, 60, 61, 110
LL for literals 63
long-running task 834
loop
body 79
continuation condition 42, 70, 71, 72, 78, 79, 87
continuation condition fails 146
counter 70
statement 42
lossless data-compression algorithm 94
lossy data-compression algorithm 94
lower_bound algorithm 620
ranges version (C++20) 592, 593
lower_bound function of associative container 536
lowercase letter 28, 112
“lowest type” 109
lvalue ("left value") 93, 131, 155, 193, 302, 436, 454, 532
*lvalue*s as *rvalue*s 93

M

m-by-*n* array 170
machine dependent 208
machine learning 222, 250
macro 111, 711
preprocessor 104
magic numbers 162
main 24
thread 775
“make your point” 119
make_heap algorithm 620
ranges version (C++20) 600
make_pair function 540
make_tuple function 681
make_unique function 428, 430, 446
mangled function name 135
manipulator 77
map associative container 533
map container class template 509
<map> header 111, 539, 541
mapped values 533
mapping in functional-style programming 179, 611
match_results class 265
suffix member function of class match_results 267
math library 103, 111
math library functions
ceil 104
cos 104
exp 104
fabs 104
floor 104
fmod 104
log 104
log10 104
pow 104
sin 104
sqrt 104
tan 104
mathematical algorithms of the standard library 574
mathematical constants 104
mathematical special functions 105
max algorithm 281, 594, 620
max heap 599
max_element algorithm 620
ranges version (C++20) 574, 576
max_size member function
container 513
string 228
maximum function 105
maximum integer value on a system 761
maximum size of a string 228, 228
mdarray container (C++23) 173
measures of central tendency 257
member function 18
automatically inlined 288
call 19
defined in a class definition 288
no arguments 288
parameter 107
member-initializer list 276, 311, 399
member object initializer 311
member selection operator (.) 291, 292, 358, 429
memberwise assignment 424
memory 16
address 192
alllocate 425, 425
consumption 373
deallocate 425
footprint of exceptions 470
leak 417, 426, 427, 431, 508
leak, preventing 429
management 103
utilization 553
memory-access violation 508
<memory> header 112, 427, 556, 621
memory-space/execution-time trade-off 552
merge
algorithm 620
algorithm, ranges version (C++20) 584, 586
member function of associative containers (C++17) 533
member function of list 529

merge symbol in the UML 47
metacharacter (regular expressions) 261
metafunction 696
return value 697
template argument 697
type 697
value 697
metaprogramming xxv
Meyer, Bertrand
design by contract 496
Microsoft xxxiv, 830
Visual C++ xxv
Microsoft C++ language documentation xxxv
Microsoft C++ Team Blog xxxv
Microsoft modularized standard library 741
Microsoft open-source C++ standard library 663
Microsoft Parallel Patterns Library concurrent containers 830
Microsoft Visual Studio Community edition xliii, 4
Microsoft Windows 82
`midpoint` algorithm 621
`milliseconds` object 761
`min` algorithm 594, 594, 620
min heap 599
`min_element` algorithm 620
ranges version (C++20) 574, 576
minimum iterator requirements 558
minimum requirements standard library algorithms 556
`miniz-cpp` library 94
`zip_file` class 94, 96
`zip_info` class 97
`minmax` algorithm 594, 595, 620
ranges version (C++20) 595
`minmax_element` algorithm 620
ranges version (C++20) 574, 576
`minus` function object 604
`mismatch` algorithm 620
ranges version (C++20) 566, 567
mismatch_result 567
`mismatch_result` for the `mismatch` algorithm 567
missing data 254
missing values 253
mixed-type expression 109
Modern C++ xxi, 2, 190
do more at compile-time 163, 626
modifiable data 757
modifiable *lvalue* 422, 436, 454
modify a constant pointer 204
modify address stored in pointer variable 204
modular architecture of this book xxv
modular standard library 740
Microsoft 740
modular standard library (C++23) 746
modules (C++20) 708, 725
building a module with partitions 735
`clang++` precompiled module (`.pcm`) file 722
`export` a block of declarations 716
`export` a declaration 716, 716, 719, 752
`export` a definition 752
`export` a `namespace` 717
`export` a `namespace` member 717
`export` followed by braces 752
`export` module 718
`export` module declaration 752
filename extension `.cpp` 719
filename extension `.cppm` 719
filename extension `.ifc` 718
filename extension `.ixx` 718, 718
filename extension `.pcm` 719
`-fmodules-ts` compiler flag (`g++`) 714
global module 752
global module fragment 752
header unit 712, 752
IFC (`.ifc`) format 752
`import` a header file 752
`import` a module 752
`import` declaration 720, 752
`import` existing header as a header unit 712
improve compilation performance 713
interface 716
linkage 753
Microsoft modularized standard library 741
modular standard library (C++23) 746
modularized standard libraries 740
modularized standard library (Microsoft) 740
`module` declaration 718, 718, 726, 753
module implementation partition unit 735
module implementation unit 726, 753
module interface 716
module interface partition unit 718, 732, 733, 734, 735
module interface partition unit `export` `module` declaration 733
module interface unit 718, 753
module interface unit (C++20 modules) 718
module interface unit compile in `clang++` 722
module interface unit compile in `g++` 721
module name 718, 753
module partition 753
module purview 718, 753
module unit 717, 753
named 726
named module 732, 753
named module purview 753
partition 732, 753
partition rules 733
precompiled module interface 753
primary module interface unit 718, 733, 734, 753
`:private` module fragment 731
`private` module fragment 753
purview 718
reachability 744
reachable declaration 753
templates 719
visibility 744
visible declaration 753
`-x c++-system-header` compiler flag (g++) 714
`modulus` function object 604
modulus operator, `%` 30
monetary formats 112
money

Boost.Multiprecision monetary library 75
Moore's law xxv, 2, 16, 16, 617, 759
most derived class 405
move 438, 447
move algorithm 620
ranges version (C++20) 586
move assignment operator xxviii, 279, 417, 431, 447, 449, 513
move constructor xxviii, 278, 417, 431, 438, 447, 448, 511, 513
move semantics xxviii, 417, 435, 513
move assignment operator 439
move constructor 438
std::move function 438, 439
move_backward algorithm 620
ranges version (C++20) 586
Move Assignable 513
multi 759
multi-core 557
architecture 618, 759
processor 17, 154, 174
systems xxix
multicore 618
multidimensional array xxvi
multidimensional array 170
multiline comment 23
multimap associative container 509, 533, 539
multipass algorithms 515
multiple inheritance 340, 340, 397, 398, 399, 400, 401
demonstration 397
diamond inheritance 402
multiple-selection statement 42
multiple-source-file program compilation and linking process 290
multiplication 30
compound assignment operator, *= 57
multiplies function object 604
multiset associative container 533
multiset container class template 509
multithreading 17, 757
condition variable 789
launch enum 814
std::async function template 814
std::future class template 814
std::packaged_task function template 815
std::shared_future class template 815
mutable (modifiable) data 757, 784
mutable data 757
mutating sequence algorithms 619, 619
mutex class (C++11) 787, 788
<mutex> header (C++11) 112, 787, 804
mutual exclusion 784, 787, 788, 827
necessary condition for deadlock 770
thread safety 757
MyArray class 430, 432, 663, 673
definition 441
definition with overloaded operators 441
test program 432

N

Nadella, Satya xxxiv
name handle 291
on an object 291
name mangling 135
to enable type-safe linkage 135
name of an array 155
named module (C++20 modules) 726, 732
named requirements 513
named return value optimization (NRVO) 437, 456
namespace keyword 719
member 720
qualifier 720
scope 124
std 24
std::chrono 761
naming conflict 315, 719
NaN (not a number) 254
narrow_cast operator 110
narrowing conversion 56, 109, 110
braced initializer 109
explicit 110
natural language processing 222
natural logarithm 104
Navigator area (Xcode) 8
Navigators
Issue 8
Project 8
NDEBUG to disable assertions 483
near container 509
negate function object 604
negative infinity (-inf) 471
nested
blocks 124
control statements 54, 55
for statement 164, 173
if...else statement 45
parentheses 31
try blocks 479
nested requirement in C++20 concepts 654, 656
nested type 510
names in containers 672
names in iterators 667
nested_exception 477
network message arrival 481
new
failure 487
failure handler 489
operator 425
<new> header 487
new operator
calls the constructor 425
placement version 425
returning nullptr on failure 489
throwing bad_alloc on failure 488
newline ('\n') escape sequence 25, 25, 216
next_permutation algorithm 621
no guarantee (of what happens when an exception occurs) 476
no preemption (necessary condition for deadlock) 770
no throw exception safety guarantee 477
node_type in an associative container 512
[[nodiscard]] attribute 292
noexcept keyword (C++11) 448, 477, 486, 497
non-const member function 307
on a const object 307

on a non-const object 307
non-type template parameter 672
non-virtual interfaces 376
nonconstant pointer to constant data 203
nonconstant pointer to non-constant data 203
noncontiguous memory layout of a deque 531
nondeterministic random numbers 114
seed 118
none_of algorithm 620
ranges version (C++20) 578, 581
non-member function to overload an operator 459
nonmodifiable *lvalue* 422
nonmodifying sequence algorithms 619, 620
non-module translation unit 725
non-parameterized stream manipulator 53
non-static member function 315, 463
non-virtual interface idiom (NVI) 338, 376
nonzero treated as true 92
not a number 84
not equal 31
not_equal_to function object 604
note in the UML 41
nothrow object 489
nothrow_t type 489
notify_all function of a std::condition_variable_any 805
notify_one function of a std::condition_variable 789, 790, 791, 799
NRVO (named return value optimization) 437, 456
nth_element algorithm 621
NULL 192
null character ('\0') 216
null in JSON 326
null pointer (0) 192, 194
null-terminated string 217
nullptr 482
on new failure 489
nullptr constant 192
number of arguments 107
number systems xxxi
<numbers> header 104
numbers in JSON 326
numeric algorithms 605, 621
<numeric> header 174, 556, 596, 621
numeric literal with many digits 63
numeric_limits class template 63
max function 761
numerical data type limits 112
NVI (non-virtual interface idiom) 376

O

$O(1)$ 550
$O(\log n)$ 551
$O(n)$ 550
$O(n^2)$ 550
object
leaves scope 298
lifetime 273
of a class 17, 19
of a derived class 352, 354
of a derived class is instantiated 349
object-oriented analysis and design (OOAD) 20
object-oriented language 20
object-oriented programming (OOP) xxv, 20, 336
object's *vtable* pointer 376
objects contain only data 290
objects natural case studies xxiv
Objects-Natural Approach xxiii, 2
octa-core processor 17
O'Dwyer, Arthur xxxix
blog xxxv
offset
from the beginning of a file 245
into a vtable 375
ofstream class 240, 241, 242
open function 242
once_flag (C++11) 816
One Definition Rule (ODR) 711, 712
one-pass algorithm 515
one-time, thread-safe initialization of an object 815
one-to-many mapping 509
relationship 539
one-to-one mapping 509, 541
online forums xxxvi
OOAD (object-oriented analysis and design) 20
OOP (object-oriented programming) 20, 336
open a file
for input 241
for output 241
that does not exist 242
open function of ofstream 242
pen source code xxxiii
community xxxiv
Microsoft C++ standard library 663
open-source class libraries 61
Open Web Application Security Project (OWASP) 327
opened 239
operand 25, 29
operating system xxi
device driver polymorphism 363
operator xxvi
-- (predecrement/post-decrement) 58
-- (prefix decrement/postfix decrement) 58
! (logical negation) 88
! (logical NOT) 90
!= (inequality) 31, 32
?: (ternary conditional) 47
() (parentheses) 31
* (multiplication) 30
* (pointer dereference or indirection) 193, 194
*= (multiplication assignment) 57
/ (division) 30
/= (division assignment) 57
&& (logical AND) 88, 89
% (remainder) 30
%= (remainder assignment) 57
+ (addition) 29, 30
++ (prefix increment/postfix increment) 58
++ (preincrement/postincrement) 58
+= (addition assignment) 57, 224
< (less-than operator) 31
<< (stream insertion) 24, 30

<= (less-than-or-equal-to) 31
= (assignment) 29, 30
-= (subtraction assignment) 57
== (equality) 31
> (greater-than) 31
>= (greater-than-or-equal-to) 31
>> (stream extraction) 29
|| (logical OR) 88, 89
address (&) 194
arrow member selection (->) 292
associativity 31
co_await 849
compound assignment 57, 59
conditional operator, ?: 47
decrement operator, -- 58
delete 425
dot (.) 37
grouping 31
logical AND, && 88, 90
logical complement, ! 90
logical negation, ! 90
logical operators 88, 90, 91
logical OR, || 88, 89
member selection (.) 291, 292
modulus, % 30
narrow_cast 110
new 425
overloading 30, 137
postfix decrement 58
postfix increment 58
precedence 30
precedence and grouping chart 35
prefix decrement 58
prefix increment 58
remainder, % 30, 858
scope resolution (::) 287
sizeof 205, 206
sizeof... 674
static_cast 52
that cannot be overloaded 423
that you do not have to overload 424
unary minus (-) 53
unary plus (+) 53
unary scope resolution (::) 133
operator
functions 423
keyword 423
operator bool stream member function 242, 244
operator overloading 182, 416
addition assignment operator (+=) 440
addition operator (+) 423, 424
binary operators 424
cast operator 454
choosing member vs. non-member functions 458
commutative operators 459
conversion operator 454
copy assignment (=) 435, 446
copy assignment operator (=) 420
decrement operators 454
equality operator (==) 435, 451
function call operator () 466
increment operators 454
inequality operator 434, 452
is not automatic 423
member vs. non-member functions 459
operator[] 453
operator+ 423
operator++ 455
operator<< 457, 458
operator= 446
operator== 451
operator>> 457
postfix increment operator 455
preincrement operator (++) 455
rules and restrictions 424
self-assignment 421
stream extraction operator >> 457
stream insertion and stream extraction operators 432, 433, 440
subscript operator 436, 453
operator[]
const version 453
non-const version 453
operator+ 423
operator<< 457, 458
operator= 446, 511
operator== 451, 566
operator>> 457
optimizing compiler 77
optional class (C++17) 191
<optional> header 113
order in which constructors and destructors are called 298, 300, 349
order in which operators are applied to their operands 144
order of evaluation 145
ordered associative container 508, 509, 509, 533
O'Reilly Online Learning xxxvi
free trial xlii
ostream class 244
seekp function 244
tellp function 245
ostream_iterator 514
ostringstream class 247, 451
out-of-bounds array elements 157
out of scope 127
out_of_bounds exception 431, 524
out_of_range exception 185, 224, 236, 492
header <stdexcept> 454
outer block 124
outer for statement 173
outliers 260
output xxvi
output iterator 515, 517, 560
output sequence 514
output stream 523
output to string in memory 112
output_iterator concept (C++20) 559, 564
output_range concept (C++20) 559, 564
outputting to strings in memory 247
overflow_error exception 492
overhead of runtime polymorphism 373
overload set in overload resolution 637, 658
overloaded function definitions 134
overloaded parentheses operator 533
overloaded stream insertion operator << 457
overloading 30, 134
concept based 652
constructor 297
function templates 651
functions 651
resolution 651
overloading << and >> 137
overloading operators 137

overload-resolution rules 453
override a function 357
`override` keyword 358, 361
OWASP (Open Web Application Security Project) 327

P

P operation on Dijkstra semaphore 827
pack a `tuple` 681
`pair` 536
`pair` class template 679
pair of braces `{}` 34
`par` execution policy of a parallel algorithm (C++17) 762, 763
`par_unseq` execution policy of a parallel algorithm (C++17) 763
parallel algorithms 618
`std::execution::par` execution policy 762, 763
`std::execution::par_unseq` executi on policy 763
`std::execution::parallel_policy` cl ass 763
`std::execution::parallel_sequenced _policy` class 763
`std::execution::seq` execution policy 763
`std::execution::sequenced_policy` cl ass 763
`std::execution::unseq` execution policy 763
`std::execution::unsequenced_policy` class 763
parallel execution 764
parallel operations 756, 756
`parallel_policy` class (C++17) 763
`parallel_sequenced_policy` class (C++17) 763
parameter 107
list 107
parameter pack 674, 683
expansion 684
variadic template 685
parameterized stream manipulator 53, 77
`quoted` 246
parameterized type 629
parentheses operator (`()`) 31
parentheses to force order of evaluation 35
partial template specialization 703, 704
`partial_sort` algorithm 621
`partial_sort_copy` algorithm 621
`partial_sum` algorithm 596, 598, 621
partition 753
`partition` algorithm 621
partition in a C++20 module 732
name 732, 733
`partition_copy` algorithm 621
`partition_point` algorithm 621
partitions building a module with 735
partitions (C++20 modules rules 733
pass-by-pointer 195
pass-by-reference 129, 191, 195, 196, 198
with a pointer parameter used to cube a variable's value 196
with pointer parameters 195
with reference parameters 130, 195
pass-by-value 129, 130, 195, 197, 203
passing arguments by value and by reference 130
`path` (C++17) 241
Paul Deitel
Full-Throttle training courses xxxvii
Live Instructor-Led Training xxxvii
payroll system using runtime polymorphism 363
`.pcm` (precompiled module) file in clang++ 719, 722
Pearson eText xxxvii
Pearson Revel xxxvii
percent sign (`%`) (remainder operator) 30
perform operations sequentially 756
performance issues with exceptions 474, 477, 482
performance tips xxiii
performing operations concurrently 756
`permutable` concept (C++20) 575
pipeline in C++20 ranges 178
placeholder in a format string 66
placeholder type 854
placement `delete` 425
placement `new` 425
platform dependency 769
Plog logging library 494
`plus` function object 604
pointer xxvii, 190, 191, 208
arithmetic 208
arithmetic, machine dependent 208
comparison 210
declared `const` 204
dereference (`*`)
operator 193, 194
expression 208
handle 291
operators `&` and `*` 194
to a function 373, 373
to an implementation 385
to dynamically allocated storage 445
to `void` (`void *`) 210
`pointer` 510
pointer-based array xxvii, 190, 191
pointer-based string 190, xxvii, 190, 191, 216
`pointer` nested type in an iterator 667
poll analysis program 167
`Poly` class template (Face-book Folly library) 409
polymorphic behavior 358
polymorphic processing 220
polymorphic video game 350
polymorphically invoking functions in a derived class 402
polymorphism 85, 334, 376
compile-time (static) 408, 410, 513, 628, 629
runtime 337
`pop`
member function of a stack 632
member function of container adapters 543
member function of `priority_queue` 546
member function of `queue` 545
member function of `stack` 543
`pop_back` member function of `list` 530
`pop_front` 527, 532, 545
`pop_heap` algorithm 620

ranges version (C++20) 602
position number 155
positive infinity (inf) 471
post contract keyword (GNU C++ early access implementation) 499
postcondition 495
violations 495
postdecrement 58
postfix decrement operator 58
postfix increment operator 58
postincrement 58, 58, 59
postincrement an iterator 517
pow function 77, 104
pow member function of class BigNumber 64
power 104
power of 2 larger than 100 47
#pragma once 284
#pragma once directive 286
pre contract keyword (GNU C++ early access implementation) 499
precedence 30, 60, 75, 144
of arithmetic operators xxvi
precedence chart 35
precedence not changed by overloading 424
precision 53, 99
precision of a floating-point value 50
precompiled module (.pcm) file (clang++) 722
precondition 495
violations 495
predecrement 58
predefined function objects 604
predicate function 292, 528, 567, 571, 573, 579, 580, 581, 582, 586, 589, 592
preemptive scheduling 769
prefix decrement operator 58
prefix increment operator 58
preincrement 58, 58, 59
preincrement operator (++) overloaded 455
“prepackaged” functions 103
preprocessor xxxi
directives 23
macro 104
state 714
prev_permutation algorithm 621
prevent memory leak 429
primary module interface unit (C++20 modules) 718, 733, 734
prime factorization 808
prime numbers
University of Tennessee Martin Prime Pages website 809
principal in an interest calculation 75
principle of least privilege 125, 203, 283, 306, 516
print spooling 776
printing
line of text with multiple statements 26
multiple lines of text with a single statement 26
priority_emplace member function of queue 546
priority_queue adapter class 546, 599, 600, 601
emplace function 546
empty function 546
pop function 546
push function 546
size function 546
top function 546
priority_queue container class template 509
privacy 148
private access specifier 274, 275
base class 407
base-class data cannot be accessed from derived class 346
inheritance 341, 406
members of a base class 341
static data member 321
:private module fragment (C++20 modules) 731
private virtual function 377
private virtual member functions 409
probability 114
procedural programming xxv
producer 757, 776, 777
producer–consumer relationship 776
profiling xxix, 759, 764
program in the general 337
program in the specific 337
program termination 302
program to an interface, not an implementation 383
programming paradigms
functional-style xxv
generic xxv
metaprogramming xxv
object-oriented xxv
procedural xxv
programming tips
C++ Core Guidelines xxiii, xxxi
C++20 modules xxiii
common programming errors xxiii
performance tips xxiii
security best practices xxiii
software engineering observations xxiii
project 5, 8
Project navigator 8
projection in C++20
std::ranges algorithms 567, 608
promise object (coroutines) 854
final_suspend member function 854
get_return_object member function 854
initial_suspend member function 854
return_value member function 854
return_void member function 854
unhandled_exception member function 854
yield_value member function 855
promotion 53
promotion rules 109
prompt 29
property injection 386
protected 405
base class 407
base class member function 406
data, avoid 406
inheritance 341, 406, 407
member of a class 405
virtual function 377
pseudorandom numbers 117
-pthread compiler flag 771
public member of a subclass 405
method 287

public access specifier 275
public base class 407
public inheritance 341, 406
public keyword 274
public services of a class 284
public static
class member 321
member function 321
pure abstract class 363, 384
pure specifier (= 0) for
a virtual function 363
pure virtual function 363
purview (C++20 modules) 718
push member function
container adapters 543
priority_queue 546
queue 545
stack 543, 632
push_back member
function vector 186, 520
push_front member
function deque 531
list 527
push_heap algorithm 620
ranges version (C++20) 601
put pointer 244

Q
quad-core processor 17
quadratic running time 550
quantifier
? 264
{n,} 264
{n, m} 264
* 263
+ 263
greedy 263
in regular expressions 262
quantum 768
questions, getting
answered xxxvi
queue 508
queue adapter class 545
back function 545
emplace function 545
empty function 545
front function 545
pop function 545
push function 545
size function 545
queue container class
template 509
<queue> header 111, 545, 546
quotation marks 24
quoted stream manipulator

R
race condition 783
radians 104
RAII (Resource Acquisition Is
Initialization) 417, 427, 431, 482,
487, 493, 776, 790
raise to a power 104
random-access
iterator 515, 516, 531
operations 517, 664
random access to elements of a
container 508
<random> header 111, 113
random integers in range 1 to
6 114
random number 117
generation xxvi
random-number generation
distribution 114
engine 114
random_access_iterator concept
(C++20) 559, 569, 570, 575, 579, 6
00, 653
random_access_range concept
(C++20) 559, 575, 579, 600, 601, 6
02
random_device random-number
source 118, 123
randomizing 117
die-rolling program 118
range (C++20) 177, 507, 514
range adaptor (C++20) 611
all 612
common 612
counted 612
drop 612, 615
drop_while 612, 615
elements 612, 617
filter 612
keys 612, 616
reverse 612, 614
split 612
take 612, 613
take_while 612, 614
transform 612, 615
values 612, 616
range-
based for statement 159, 224
with initializer 161
range checking 224, 431
range concept (C++20) 559
range factory 613
range factory (C++20)
iota 613
iota for an infinite range 613
range of elements 525
range-v3 project 622
range variable 160, 172
ranges concepts (C++20) 559
<ranges> header (C++20) 113, 177
ranges library (C++20) 177, 253
rapidcsv header-only library 251
Document class 252
GetColumn member function of
class Document 252, 254
GetRowCount member function of
class Document 255
rapidcsv library 223
RapidJSON library 251
raw data 246
raw string literal 249
Raz, Saar xxxix
rbegin
member function of
containers 512
member function of vector 522
reachability (C++20
modules) 744
read 665
read and print a sequential
file 243
read data sequentially from a
file 243
readers and writers
problem 804
std::shared_mutex class
(C++11) 804
ready thread state 768
real-time systems xxi
record 240
recursion xxvi, 139, 146, 147
step 140, 144
recursive call 140, 144
factorial 142
function 139
solution 147
reddit.com/r/cpp/ xxxvi
reduce
algorithm 596, 597, 621
parallel algorithm (C++17) 766

reduction 163, 174, 175, 180, 597, 685
refactor 338
payroll example 384
reference 510
argument 195
parameter 129, 129
to a constant 131
to a local variable 131
to an int 129
to private data 302
reference nested type in an iterator 667
<regex> header 261, 265
regex library
cmatch 265
match_results class 265
regex_constants 266
regex_match function 261
regex_replace function 265
regex_search algorithm 265
regex_search function 265
smatch 265
regular expression 103, 247, 259, 266
? quantifier 264
[] character class 262
{n,} quantifier 264
{n,m} quantifier 264
* quantifier 263
\ metacharacter 261
\d character clas 262
\D character class 262
\d character class 262
\S character class 262
\s character class 262
\W character class 262
\w character class 262
+ quantifier 263
caret (^) metacharacter 263
case insensitive 261
case sensitive 261
character class 261, 262
ECMAScript grammar 260
escape sequence 262
flavors 260
grammars 260
metacharacter 261
regex_constants::icase 266
search pattern 260
validating data 260
relational function object 604
operator 31, 32
release
a lock 787, 790
a semaphore 827
release member function of std::binary_semaphore 829, 830
relinquish the processor (yield) 819
remainder after integer division 30
remainder compound assignment operator, %= 57
remainder operator (%) 30, 30, 31, 858
remove algorithm 620
ranges version (C++20) 568, 569
remove member function of list 531
remove_copy algorithm 620
ranges version (C++20) 568, 570
remove_copy_if algorithm 620
ranges version (C++20) 568, 572
remove_if algorithm 620
ranges version (C++20) 568, 571
remove_prefix member function of string_view 238
remove_suffix member function of string_view 238
rend
member function of containers 512
member function of vector 522
repetition
counter controlled 52
sentinel controlled 50, 51, 52
repetition statement 42
do...while 42
for 42
while 42, 49, 52
replace == operator with = 92
replace algorithm 572, 620
ranges version (C++20) 572, 572
replace member function of class string 232, 233
replace_copy algorithm 620
ranges version (C++20) 572, 573
replace_copy_if algorithm 620
ranges version (C++20) 572, 574
replace_if algorithm 620
ranges version (C++20) 572, 573
representational 78
representational error 78
representational error in floating point 78
reproducibility xxxiv
request_stop member function of a std::jthread 807, 808
requirements 20
requires clause (C++20) 640
requires expression (C++20) 654
reserve member function of class string 230
reserved word 42
false 44
true 44
reset 547
resize member function of class string 230
Resource Acquisition Is Initialization (RAII) 417, 427, 482, 487, 493
resource leak 429, 475, 493
resource sharing 769
result (concurrencpp) 841
rethrow an exception 477
return a value 24
Return key 29
return statement 25, 108, 140
return_value function of a coroutine promise object 854
return_void function of a coroutine promise object 854
returning a reference from a function 131
returning a reference to a private data member 302
reusable software components 17
reuse 18, 37, 272
Revel (Pearson) xxxvii
reverse algorithm 620
ranges version (C++20) 584, 587
reverse range adaptor (C++20) 612, 614
reverse_copy algorithm 620
ranges version (C++20) 588, 589
reverse_iterator 510, 512, 516, 522
rfind member function of class string 231
right align > (C++20 text formatting) 99, 100
right brace (}) 24, 49, 52

right fold
binary 689
unary 688
right operand 25
right shift operator (>>) 416
right stream manipulator 77
right value 93
rightmost (trailing)
arguments 133
robust application 469
rolling dice 115, 119
rotate algorithm 620
rotate_copy algorithm 620
round a floating-point number
for display purposes 54
round-robin scheduling 769
rounding numbers 54, 78, 104
rows 170
RSA Public-Key Cryptography
algorithm 808
RTTI (runtime type
information) 409
Rule of Five (for special
member functions) 444
Rule of Five defaults 444
Rule of Zero (for special
member functions) 279, 444, 665
rules of operator precedence 30
run-length encoding 94
running state 768
runtime (concurrencpp) 841, 843
runtime concept idiom 408
private virtual member
functions 409
runtime polymorphism 337
using class hierarchies 376
with virtual functions 358
runtime type information
(RTTI) 409
runtime_error class 472, 480, 491,
492
what function 476
rvalue ("right value") 93
rvalue (“right value”) 131
rvalue reference
(&&) 435, 438, 439, 447, 448, 449

S
SalariedEmployee class
header 367
implementation file 368
same_as concept (C++20) 649
sample algorithm 620
savings account 75
schedule a task to execute 844
scheduling threads 769
scientific notation 53
scope 124, 719
class 124
example 125
file 125
function 124
function parameter 124
namespace 124
scope of a variable 73
scope resolution operator
(::) 121, 124, 287, 321, 399, 634,
720
scoped enumeration (enum
class) 120
scoped_lock class (C++11) 791
scraping 260
screen 23
screen-manager program 350
scrutinize data 287
search algorithm 620
search pattern (regular
expressions) 260
search_n algorithm 620
searching 508, 578
arrays xxvi, 168
second data member of pair 536
secondary storage 16
secondary storage device 222
security 148
best practices xxiii
flaws 157
seed
nondeterministic 118
the random-number
generator 117, 118
seek
direction 245
get 244
put 244
seekg function of istream 244
seekp function of ostream 244
selection 43
selection statement xxvi, 41, 42
if 42, 44
if...else 42, 44, 45, 52
switch 42, 84
with initializer 85
self-assignment 449
in operator overloading 421
self-driving car 469
semaphore 826
acquire 827
release 827
<semaphore> header
(C++20) 113, 827
semicolon (;) 24, 34
send a message to an object 19
sentinel (C++20 ranges) 525
sentinel-controlled
iteration xxvi, 51, 52
sentinel value 50, 52
separate interface from
implementation 284
seq execution policy (C++17) 763
sequence 41, 43, 170, 514
sequence
container 508, 508, 516, 518, 524
, 528, 545
back function 524
empty function 525
front function 524
insert function 524
sequence of random
numbers 117
sequenced_policy class
(C++17) 763
sequential file 240, 243, 246, 247
serialization xxvii
avoid language native
serialization 327
pure data formats 327
security 327
serializing data 326
set associative
container 533, 537
set container class template 509
set function
validate data 279
<set> header 111, 533, 537
set_new_handler function 487, 489,
490
set operations of the standard
library 590
set_difference algorithm 620
ranges version (C++20) 589, 591
set_intersection algorithm 620
ranges version (C++20) 589, 591
set_symmetric_difference algorith
m 620
ranges version (C++20) 589, 591

set_union algorithm 620
ranges version (C++20) 592
setprecision stream manipulator 53
setw parameterized stream manipulator 77
SFINAE (substitution failure is not an error) 410, 658
obviated by C++20 concepts 410
shadow 315
shallow copy 445
Shape class hierarchy 340
share data 757
shared buffer 777
shared mutable data 757, 784
shared_lock class (C++11) 804
shared_mutex class (C++11) 804
<shared_mutex> header 112
shared_ptr class 428
shell prompt on Linux 4
shift_left algorithm 620
shift_right algorithm 620
shifted, scaled integers 115
short-circuit evaluation 90, 642
shrink_to_fit member function of classes vector and deque 522
shuffle algorithm 620
ranges version (C++20) 574, 575
side effect 129
of an expression 125, 129, 145
signal
a latch 820
operation on semaphore 827
signal value 50
signature 108, 135
of overloaded prefix and postfix increment operators 455
overridding a base-class virtual function 357
SIMD (single instruction, multiple data) instructions 759
simple condition 88
simple requirement in C++20 concepts 654, 654
simulation techniques xxvi
sin function 104
sine 104
single-argument constructor 464, 465
single-entry/single-exit control statements 43
single
inheritance 340, 399, 401, 402
single instruction, multiple data (SIMD) instruction 759
single-line comment 23
single-precision floating-point number 77
single quote 25
single quote (') 216
single-selection statement 42
if 44
single-threaded application 757
single-use gateway 820
singly linked list 508, 527
six-sided die 114
size
of a string 227
of a vector 519
of an array 205
size global function 181
size member function
array 155
containers 513
initializer_list 443
priority_queue 546
queue 545
stack 543, 632
string_view 239
vector 181
size_t type 157, 205
size_type 510
sizeof operator 205, 206
applied to an array name returns the number of bytes in the array 206
used to determine standard data type sizes 206
sizeof... operator 674
sleep interval 768
sleep_for function of the std::this_thread namespace (C++11) 773
sleep_until function of the std::this_thread namespace (C++11) 773
sleeping thread 768
slicing (logic error) 475
small circles in the UML 41
smart pointer 427
make_unique function template 428, 430
unique_ptr 431
smatch 265
str member function 267
software engineering
information hiding 274
observations xxiii
reuse 37, 272, 283
separate interface from implementation 284
software reuse 103, 630
solid circle in the UML 41
solid circle surrounded by a hollow circle in the UML 41
solution 5
Solution Explorer 5
Solution Explorer in Visual Studio Community Edition 5
sort algorithm 168, 169, 459, 620, 759
ranges version (C++20) 578, 579, 609, 610
sort member function of list 528
sort_heap algorithm 620
ranges version (C++20) 601
sorting 508, 578
arrays xxvi
arrays 168
order 580, 586
related algorithms 619
strings 112
space–time trade-off 521
spaceship operator (<=>) xxviii, 459
span class template (C++20) 191, 210
back member function 214
first member function 215
front member function 214
last member function 215
subspan member function 215
 header (C++20) 113, 191, 210
spdlog logging library 494
special characters 28
special member functions xxviii, 278, 417, 431
constructor 275, 278
containers 511
copy assignment operator xxviii, 278, 417, 420, 446

copy
constructor xxviii, 278, 417, 446
destructor xxviii, 279, 298, 417
move assignment
operator xxviii, 279, 417
move
constructor xxviii, 278, 417
remove with = delete 444
Rule of Five 444
Rule of Zero 444
specialized memory
algorithms 621
specialized memory
operations 619
spiral 143
splice member function
of list 528
splice_after member function of
class template forward_list 528
split range adaptor (C++20) 612
spooling 777
spurious wakeup 789
sqrt function
of <cmath> header 104
square function 110
square root 104
<sstream> header 112, 247, 247
stable_partition algorithm 621
stable_sort algorithm 621
stack 507
stack adapter class 509, 543
emplace function 543
empty function 543
pop function 543
push function 543
size function 543
top function 543
Stack class
template 629, 632, 633
stack data structure 629, 630
<stack> header 111, 543
stack overflow 140
stack unwinding 476, 479, 487
stackful coroutine 840
stackless coroutine 840
StackOverflow xxxiii, xxxvi
stackoverflow.com xxxvi
stale value 784
Standard C++ Foundation xxxv
standard C++20 concepts by
header 642
standard concepts (C++20) 640
standard data type sizes 206
standard document (C++) xxxv
standard exception classes 492
bad_alloc 487, 491
bad_cast 491
bad_typeid 491
exception 491, 491
invalid_argument 492
length_error 492
logic_error 491
out_of_range 492
overflow_error 492
runtime_error 472, 480, 491, 492
underflow_error 492
standard input stream object
(cin) 240
standard library 103
class string 35, 418
deque class template 532
exception hierarchy 490
headers 112
list class template 527
map class template 541
multimap class template 539
multiset class template 534
priority_queue adapter class 547
queue adapter class
templates 545
set class template 538
stack adapter class 543
vector class template 519
standard library algorithms
minimum requirements 556
standard library exception
filesystem_error 494
standard library exception
hierarchy 490
standard output object (cout) 24
standard output stream object
(cout) 240
Standard Template Library
(STL) 506
Start Window 4
Start Window in Visual Studio
Community Edition 4
starts_with member function of
class string (C++20) 38
starts_with member function
of string_view 239
starvation 769
state dependent 777
statement 24
break 83, 86
continue 86
control statement 41, 43
control-statement nesting 43
control-statement stacking 43
do...while 42, 78
double selection 42
empty 46
for 42, 71, 75, 77
if 31, 42, 44
if...else 42, 44, 45, 52
iteration 41, 47
looping 42
multiple selection 42
nested control statements 54
nested if...else 45
repetition 42
return 25
selection 41, 42
single selection 42
spread over several lines 34
switch 42, 80, 84
throw 287
try 185
while 42, 47, 49, 52, 70
static binding 358
static code analysis tools xxxii
clang-tidy xxxii, xlviii
cppcheck xxxii, xlviii
static data member 320, 321
save storage 320
tracking the number of objects
of a class 323
static keyword 123, 124
static local object 299, 301, 302
static local
variable 125, 127, 564
thread-safe
initialization 815, 816
static member 321
static member function 321
static
polymorphism 408, 410, 629
static polymorphism (compile-
time) 628
static_assert declaration 659
static_cast operator 52, 60, 92
statistics
measures of central
tendency 257
std namespace 24
std::add_const metafunction 703

std::advance function 652
std::as_const function (C++17) 676
std::async (C++11) 808
std::async function template 814
std::atomic class template 817
std::atomic type 817
std::atomic_ref class template (C++20) 817, 820
std::barrier (C++20) 820, 823
std::binary_semaphore (C++20) 827
acquire member function 829, 830
release member function 829, 830
std::call_once (C++11) 816
std::call_oncedefault para font> (C++11) 816
std::chrono namespace 761
std::chrono::duration class 761
std::cin 29
std::condition_variable class
notify_one function 789, 790, 791, 799
std::condition_variable class (C++11) 787
std::condition_variable_any class (C++11) 805
notify_all function 805
std::counting_semaphore (C++20) 827
std::cout 24
std::distance function 652
std::execution::par execution policy (C++17) 762, 762, 763
std::execution::par_unseq executi on policy (C++17) 763
std::execution::parallel_policy cl ass (C++17) 763
std::execution::parallel_sequenced _policy class (C++17) 763
std::execution::seq execution policy (C++17) 763
std::execution::sequenced_policy cl ass (C++17) 763
std::execution::unseq execution policy (C++17) 763
std::execution::unsequenced_policy class (C++17) 763
std::floating_point concept 641, 648
std::future class template 814
std::hash 533
std::initializer_list class template 443
std::integral concept 641, 648
std::invoke function 610
std::jthread (C++20)
join function 775
request_stop function 807, 808
std::latch (C++20) 820
count_down member function 821
wait member function 821, 821
std::launch enum
async 814
deferred 814
std::lock_guard class (C++11) 791
std::move function 438, 439
std::mutex class (C++11) 787, 788
std::numeric_limits::max() 761
std::once_flag (C++11) 816
std::optional class (C++17) 191
std::packaged_task function template 815
std::promise (C++11) 814
std::ranges namespace 507, 556
all_of algorithm 578, 580
any_of algorithm 578
std::ranges namespace (C++20) 525, 560, 561, 563, 566, 568, 572, 574, 578, 582, 584, 588, 589, 592, 594, 599
std::ranges::count_if algorithm (C++20) 257, 258
std::ranges::distance algorithm 663
std::reverse_iterator iterator adatpor 672
std::same_as concept (C++20) 649
std::scoped_lock class (C++17) 791
std::shared_future class template 815
std::shared_lock class (C++11) 804
std::shared_mutex class (C++11) 804
std::size global function 181
std::stop_callback (C++20) 808
std::stop_source for cooperative cancellation (C++20) 807
std::stop_token for cooperative cancellation (C++20) 807
stop_requested function 807
std::string_literals 168
std::this_thread namespace
yield function 819
std::this_-thread::get_id function (C++11) 772
std::this_-thread::sleep_for function (C++11) 773
std::this_-thread::sleep_until function (C++11) 773
std::thread class (C++11) 771
std::thread::id 772
std::unique_lock class (C++11) 788, 789
std::variant class template 391
for runtime polymorphism 391
std::visit standard library function 391, 395, 396
std:jthread (C++20) 771, 771, 776
std.core in the Microsoft modularized standard library 740
std.filesystem in the Microsoft modularized standard library 740
std.memory in the Microsoft modularized standard library 740
std.regex in the Microsoft modularized standard library 740
std.threading in the Microsoft modularized standard library 741
<stdexcept> header 112, 472, 491
out_of_range 454
steady_clock class 761
sticky setting 54, 77
STL (Standard Template Library) 506
stod function 235
stof function 235
stoi function 235
stol function 235
stold function 235
stoll function 235
stop_requested member function of a std::stop_token 807
<stop_token> header 113
<stop_token> header (C++20) 805, 820
stoul function 235

stoull function 235
str member function of an smatch 267
str member function of class ostringstream 247, 248
stream extraction operator >> 29, 34, 137, 416, 456
stream input/output 23
stream insertion operator << ("put to") 416
stream insertion operator << 25, 29, 137, 243, 456
stream manipulator 77
boolalpha 37
quoted 246
stream manipulators 53
fixed 53
left 77
right 77
setprecision 53
setw 77
stream of characters 24
streaming 757
<string> header 112
string 168, 616
C style 190
pointer based 190
processing xxvii
string iterators 568
string built-in type in JSON 326
string class 35, 36, 112, 273, 417, 418, 509
assignment 223
assignment and concatenation 223
at member function 422
concatenation 223
empty member function 38, 419
ends_with member function (C++20) 38
find functions 230
find member function 230
insert functions 234
insert member function 234
length member function 37
starts_with member function (C++20) 38
subscript operator [] 224
substr member function 421
string concatenation 38, 685
string concatenation assignment 420
string formatting 65
C++20 65, 66, 67
<string> header 37
string literal 24
raw string literal 249
string object
literal 168, 420, 616, 635
string of characters 24
string processing 103
string stream processing 247
string_literals 168
string_view class
(C++17) 190, 236, 274
find member function 239
remove_prefix member function 238
remove_suffix member function 238
size member function 239
starts_with member function 239
<string_view> header (C++17) 236
string::npos 231
strings as full-fledged objects 216
strong encapsulation 722, 743
strong exception guarantee 446
copy-and-swap idiom 477
strong exception safety guarantee 477
Stroustrup, Bjarne website xxxv
struct 324
structured binding (C++17) 595
unpack elements 577
structured binding declaration 577
structured programming 40, 88
student-poll-analysis program 167
subclass 336
submit function of a concurrencpp executor 844
subobject of a base class 402
subproblem 140
subscript 155
subscript operator 532
map 541, 542
subscripted name used as an *rvalue* 436
subscripting 531
subspan member function of span class template (C++20) 215
substitution cipher 148
substr 227
substr member function of class string 226, 421
substring of a string 226
subtract one pointer from another 208
subtraction 30, 31
subtraction compound assignment operator, -= 57
sufficient conditions for deadlock 770
suffix member function of class match_results 267
sum of the elements of an array 163, 174
superclass 336
survey 166, 168
suspend a coroutine's execution 839
suspend_always (coroutines) 854
suspend_never (coroutines) 854
suspension point (coroutines) 855
Sutter, Herb 376
blog 482, 496
ISO C++ Convener 496
Sutter's Mill Blog xxxv
swap member function of class unique_ptr 459
standard library function 459
swap algorithm 583
swap member function of containers 513
of list 530
swap member function of class string 227
swap_ranges algorithm 582, 620
ranges version (C++20) 583, 584
swapping strings 227
switch logic 85
switch multiple-selection statement 42, 80, 84
case label 83
controlling expression 83
default case 83, 84
switch with initializer 85
synchronization 784, 785

synchronization point 820
synchronized block of code 787
synchronized threads 757
synchronous error 481
syntax coloring conventions in this book xxxiii
system_clock class 761

T

Tab key 24
tab stop 25
table of values 170
tabular format 157
tag dispatch 411, 658
obviated by C++20 concepts 411
tail of a queue 508
tails 114
take range adaptor (C++20) 612, 613
take_while range adaptor (C++20) 612, 614
tan function 104
tangent 104
task (concurrencpp) 841
task for asynchronous operations 836
tellg function of istream 245
tellp function of ostream 245
template definition 138
template function 138
template header 630
template instantiations 137
template keyword 137, 630
template metaprogramming xxv
template metaprogramming (TMP) xxv, 628, 693
metafunction 696
Turing complete 694
type metafunction 697
value metafunction 697
template parameter 630
template parameter list 137
templated lambda (C++20) 636
templated lambda expression 634
templates compile-time code generation 410
constraints 411
deduction guide 673
default type argument for a type parameter 678
defining in C++20 modules 719
partial specialization 703
requirements for a type 410
type argument 630
variable template 678
temporary value 52, 110
terminate a program 489
terminate normally 242
terminate standard library function 480
terminate successfully 25
terminated state 769
terminating condition 141
terminating right brace (}) of a block 124
termination
abort function 480, 489
exit function 489
terminate function 480
termination condition 157
termination housekeeping 298
termination model of exception handling 475
termination test 146
ternary conditional operator (?:) 145
ternary operator 47
test 548
test characters 112
text editor 243
text formatting 98
C++20 65, 66, 67
format specifier 99
text-printing program 23
the cloud 326
this pointer 315, 316, 324, 447, 450
used explicitly 315
used implicitly and explicitly to access members of an object 315
thread 757
exception 771
of execution 757
scheduling 768, 782
state 767
synchronization 784
thread class (C++11) 771
thread-coordination primitives 816
thread-coordination types 820
<thread> header 112, 771
thread launch policy 814
thread-local storage thread safe 758
thread pool 841
thread safe 757, 783, 787
one-time initialization 815
atomic type 757
immutable data 757
linked data structures 819
mutual exclusion 757
thread local storage 758
thread scheduler 769
thread states
blocked 769
born 768
ready 768
running 768
terminated 769
timed waiting 768
waiting 768
thread synchronization coordination types 820
thread_executor (concurrencpp) 844
thread_local storage class (C++11) 758
thread_pool_executor (concurrencpp) 841, 844
thread::id 772
C++20) xxviii, 113, 279, 459, 460, 511
three-way comparison operator (<=> xxviii, 113, 279, 459, 460, 511
throw 476, 491
standard exceptions 491
throw an exception 184, 185, 287, 287, 474
from a constructor 484
throw point 474, 480
tightly coupled 383
tilde character (~) 298
time and date utilities 761
Time class
constructor with default arguments 293
definition 285
definition modified to enable cascaded member-function calls 317
member-function definitions 286

member-function definitions, including a constructor that takes arguments 294
timed waiting thread state 768
timer for performing a task in the future 836
times 103
timeslice 768, 769
Titanic disaster dataset 223, 253
`tl::generator` class template (generator library) 837
TMP (template metaprogramming) 628, 693
`to_array` function of header `<array>` (C++20) 191, 201
`to_string` function 235
token 246
`top` member function of `priority_queue` 546
of `stack` 543
`top` member function of a stack 631
top of a stack 507
topical xxi
trailing `requires` clause 649
trailing return types 563
transaction processing 539
`transform` algorithm 620
ranges version (C++20) 574
`transform` range adaptor (C++20) 612, 615
`transform_exclusive_scan` algorithm 621
`transform_exclusive_scan` parallel algorithm (C++17) 766
`transform_inclusive_scan` algorithm 621
`transform_inclusive_scan` parallel algorithm (C++17) 766
`transform_reduce` algorithm 621
`transform_reduce` parallel algorithm (C++17) 766
transforming data 260
transition arrow in the UML 41
transition from the preprocessor to modules 712
translation look-aside buffers (TLBs) 553
translation unit 679, 710, 713, 732
non-module 725
part of a module 717
treat warnings as errors 131
trigonometric cosine 104
trigonometric sine 104
trigonometric tangent 104
triple indirection 373
trivially copyable type 819, 820
`true` 32, 44
truncate 30, 241
truncate fractional part of a calculation 50
truncate fractional part of a `double` 109
truth tables for logical operators 88, 89
`try` block 185, 477, 480
expiration 475
nested 479
`try` statement 185
`tuple`
pack 681
unpacking 681
`tuple` class template 679
getting a tuple member 681
`make_tuple` function 681
`<tuple>` header (C++11) 111, 679
Turing complete 694
two-dimensional `array` 170
type alias 680
type argument 430, 604, 630
type category 648
type erasure 409
type-erasure-based runtime polymorphism 409
type metafunction 697
predefine 704
type name, alias 394
type of the `this` pointer 315
type parameter 137, 138, 630
type requirement in C++20 concepts 640, 654, 655
type-safe linkage 135
type-safe `union` 394
type trait 628
`is_base_of` 699
`value` member 646
`<type_traits>` header 701, 644
`typeid` 491
`<typeinfo>` header 112
`typename` keyword 137, 630
`typename...` in a variadic template 683

U

Ubuntu Linux 7
in the Windows Subsystem for Linux 7
UML (Unified Modeling Language)
activity diagram 41, 41, 47, 79
arrow 41
diamond 44
dotted line 42
final state 41
guard condition 44
merge symbol 47
note 41
solid circle 41
solid circle surrounded by a hollow circle 41
UML class diagram 340
unary left fold 686, 688
unary minus (`-`) operator 53
unary operator 53, 90, 193
unary operator overload 424
unary plus (`+`) operator 53
unary predicate function 571, 573
unary right fold 686, 688
unary scope resolution operator (`::`) 133
uncaught exception 479, 480
unconstrained function template 637
undefined behavior 63, 194, 426, 445
division by zero 471
undefined value 278
`underflow_error` exception 492
underlying data structure 546
underscore (_) 28
`unhandled_exception` function of a coroutine promise object 854
Unicode character set 85
`uniform_int_distribution` 114
unincremented copy of an object 455
`union` 394
`unique` algorithm 620
ranges version (C++20) 584, 586
unique keys 533, 537, 541
`unique` member function of `list` 530
`unique_copy` algorithm 620

ranges version (C++20) 588, 589
unique_lock class (C++11) 788, 789
unlock function 790
unique_ptr 431
unique_ptr class
(C++11) 428, 428, 431
built-in array 430
create with make_unique function
template 446
get member function 444
swap member function 459
universal-time format 287
University of Tennessee Martin
Prime Pages website 809
UNIX 82, 242
unlock function of
a unique_lock 790
unordered associative
containers 508, 509, 511, 533
unordered_map associative
container class
template 509, 533, 541
<unordered_map> header 111, 539,
541
unordered_multimap associative
container class
template 509, 533, 539
unordered_multiset associative
container class
template 509, 533
unordered_set associative
container class
template 509, 533, 537
<unordered_set> header 111, 533,
537
unpack elements (C++17
structured binding 577
unpacking a tuple 681
Unruh, Erwin 694
unseq execution policy
(C++17) 763
unsequenced_policy class
(C++17) 763
unsigned char data type 110
unsigned data type 110
unsigned int data type 110
unsigned integer types 109
unsigned long data type 110
unsigned long int data type 110
unsigned long long data type 110
unsigned long long int data
type 110
unsigned short data type 110
unsigned short int data type 110
unwinding the function call
stack 479
update records in place 246
upper_bound algorithm 620
ranges version (C++20) 592, 593
upper_bound function of
associative container 536
uppercase letter 28, 112
user-defined function 105
user-defined
type 120, 121, 272, 462
using a dynamically
allocated ostringstream object 24
7
using a function template 137
using arrays instead of switch 165
using declaration 33, 720
in headers 274
using declaration to create an
alias for a type 394, 680
using directive 33, 720
in headers 274
using enum statement 124
using function swap to swap
two strings 227
using standard library functions
to perform a heapsort 600
using virtual base classes 403
Utilities area (Xcode) 8, 9
utility function 292
<utility> header 112
exchange function 448

V

V operation on semaphore 827
validate a first name 262
validate data in
a *set* function 279
validating data (regular
expressions) 260
value
initialization 159, 199, 278, 325,
426, 682
memory 443
objects 426
rules 426
value member of a type-trait
class 646
value metafunction 697
value of an array element 155
value_type 510
nested type in an iterator 667
values range adaptor
(C++20) 612, 616
variable 27
variable scope 73
variable template
(C++14) 646, 678
variadic function template 686
compile-time recursion 682
typename... 683
variadic template 628, 674, 679
parameter pack 674, 683
sizeof... operator 674
variadic template parameter
pack 685
<variant> header 391
variant standard library class
template 391
vector 631
capacity 519
vector class 180, 482
vector class
template 154, 508, 519
capacity function 519
crbegin function 522
crend function 522
erase member function 569
push_back function 520
push_back member function 186
push_front function 520
rbegin function 522
rend function 522
shrink_to_fit member
function 522
vector class template element-
manipulation functions 523
vector hardware operations 759
<vector> header 111, 180
vector mathematics 618, 759
vectorized execution 764
version control tools xxxiv
video streaming 795
videos
C++20 Fundamentals
LiveLessons xxxvii
view (C++20) 177, 507, 611
all range adaptor 612
common range adaptor 612
composable 177
composing 611
counted range adaptor 612

`drop` range adaptor 612, **615**
`drop_while` range adaptor 612, **615**
`elements` range adaptor 612, 617
`filter` range adaptor 612
`iota` range factory 613
`iota` range factory for an infinite range **613**
`keys` range adaptor 612, **616**
`reverse` range adaptor 612, **614**
`split` range adaptor 612
`take` range adaptor 612, **613**
`take_while` range adaptor 612, **614**
`transform` range adaptor 612, 615
`values` range adaptor 612, **616**
view into a container 210
`viewable_range` (C++20) **611**
views of contiguous container elements 210
`views::filter` **178**, 179
`views::iota` **178**
Vigenère secret key cipher **147**, 148, 149, 150
violation handler (contracts) **503**
default **500**
`virtual` base class 402, 403
`virtual` destructor **361**
`virtual` function 337, **357**, 357, 373, 375, 402
as an internal implementation detail 382, 412
call 375
call illustrated 374
overhead 373
`private` 377, 409
`protected` 377
table (*vtable*) **373**
"under the hood" 373
`virtual` inheritance **403**
virtual memory 488, 490
Virtuality (paper) 376
visibility (C++20 modules) **744**
`visit` standard library function **391**, 395, 396
visitor pattern **411**
Visual C++ xxii
Visual C++ compiler xliv
Visual Studio Community Edition xliii, 4
Command Prompt window 6
Create a New Project dialog 5
Create a New Project-Configure your new project 5
Empty Project template 5
Solution Explorer 5
Start Window 4
`void *` **210**
`void` return type **107**, 108, 109
volume of a cube 128
vtable **373**, 376
vtable pointer 376
vtable pointer 376

W

wait-for (necessary condition for deadlock) 770
`wait` function of a `condition_variable` **789**
`wait` member function of a `std::latch` **821**
wait operation on semaphore 827
waiting thread 790
state 768
"walk off" either end of an array 430
`-Wall` GNU g++ compiler flag 93
warnings
treat as errors 131
`weak_ptr` class (C++11) 428
`weakly_incrementable` concept (C++20) 561
web service 326
Welcome to Xcode window 8
`what` virtual function of class `exception` **186**, 475, 476, 488
`when_all` function (`concurrencpp` library) **848**
`when_any` function (`concurrencpp` library) **849**
`while` iteration statement 42, **47**, 49, 52, 70
whitespace characters **23**, 24
whole number 27
Williams, Anthony xxxix
Windows 82
Windows Subsystem for Linux (WSL) xxvi, **xlv**, 7
word character **262**
`worker_thread_executor` (concurrencpp) **845**
workflow **41**
workspace window 8

X

`-x c++-system-header` compiler flag (g++) **714**
x86-64 gcc (contracts) 498
Xcode xliii
Debug area 9
Editor area 8, 9
Navigator area 8
Utilities area 8, 9
Welcome to Xcode window 8
Xcode navigators
Issue 8
Project 8
Xcode on Mac OS X 4
XML (eXtensible Markup Language) 326
xvalue (expiring value) **438**

Y

`yield` function of namespace `std::this_thread` 819
`yield` the processor 819
`yield_value` function of a coroutine promise object **855**
Yoda condition 93

Z

zero-overhead principle of C++ features 470
ZIP file format **94**
`zip_file` class from the `minixcpp` library 94, 96
`zip_info` class from the `minixcpp` library 97